CS51 Discussion and Project Book

CS51 Course Staff

May 26, 2004
## Contents

### 1 The Structure of LISP

1.1 Introduction to LISP ........................................ 1
  1.1.1 S-Expressions in LISP ................................ 1
  1.1.2 LISP Interpretation and Compilation ................... 3
  1.1.3 Introduction by Analogy with C (or Pascal) .......... 6

1.2 Internal Representation of LISP Values .................... 11
  1.2.1 Internal Representation of LISP Numbers ............... 13
  1.2.2 Internal Representation of LISP Symbols ............... 16
  1.2.3 Internal Representation of “Ordinary” Lists ............ 20
  1.2.4 Internal Representation of Dotted-Pair Forms .......... 22
  1.2.5 Internal Representation of LISP Arrays ................. 24
  1.2.6 LISP Equality Testing Functions ...................... 25

1.3 Summary of LISP ............................................ 27
  1.3.1 LISP Variables ....................................... 27
      1.3.1.1 Non-Special Symbols and Lexical Binding ....... 28
      1.3.1.2 Special Symbols and Dynamic Binding .......... 29
  1.3.2 LISP Functions, Macros, and Special Forms ........... 31
      1.3.3 How LISP Evaluates S-Expressions ................. 34
  1.3.4 Built-in Functions, Macros, and Special Forms ......... 36
      1.3.4.1 List Structure ................................ 36
      1.3.4.2 Symbol Structure ................................ 37
      1.3.4.3 Array Structure ................................ 38
      1.3.4.4 Local Variables ................................ 39
      1.3.4.5 Global Variables ................................ 40
      1.3.4.6 Defining and Using Functions .................. 40
3.4 An Example Using Pattern-Matching ........................................... 123
3.5 Pattern Matching Projects ....................................................... 126
  3.5.1 Project: Sentence Generator ............................................... 126
    3.5.1.1 A Simple Grammar .................................................. 126
    3.5.1.2 Representing Grammars in LISP .................................. 129
    3.5.1.3 Implementing a Sentence Generator .............................. 129
    3.5.1.4 Optional Additions .................................................. 130
  3.5.2 Project: Animals ............................................................ 130
    3.5.2.1 The Game of Animals .............................................. 130
    3.5.2.2 Optional Additions .................................................. 134

4 Searching Strategies 135
  4.1 Searching a Network ........................................................ 135
  4.2 Depth-First Search .......................................................... 137
  4.3 Breadth-First Search ....................................................... 142
  4.4 Tree Representation of Searches .......................................... 144
  4.5 Depth-First Search vs. Breadth-First Search .......................... 145
  4.6 State-Space Search ......................................................... 146
  4.7 Other Searching Strategies .............................................. 155
  4.8 Search Strategy Projects ................................................... 160
    4.8.1 The Eight-Puzzle ...................................................... 160
      4.8.1.1 Optional Additions .............................................. 161
    4.8.2 Shortest Paths in Graphs ............................................. 162
      4.8.2.1 Weighted Graphs .................................................. 162
      4.8.2.2 A* Algorithm for a Road Map ................................ 163
      4.8.2.3 Implementing the A* Algorithm ............................... 165
    4.8.3 Heuristic Maze Searches ............................................. 168
      4.8.3.1 Representing Mazes .............................................. 168
      4.8.3.2 Applying Depth-First and Breadth-First Searches ........... 169
      4.8.3.3 Implementing Heuristic Strategies ............................ 169
      4.8.3.4 Optional Additions .............................................. 170

5 Propositional Calculus and Theorem-Proving 171
  5.1 Propositional Variables and Boolean Operators ........................ 171
  5.2 Theorems and Satisfiability .............................................. 173
  5.3 Transformations of Formulas ............................................. 175
  5.4 Representing and Simplifying Formulas in LISP ........................ 178
  5.5 Theorem-Proving: The Resolution Method .............................. 189
CONTENTS

5.6 Propositional Calculus Projects ............................................. 193
  5.6.1 CNF Conversion (Revisited) Project ............................... 193
    5.6.1.1 An Alternative CNF-Conversion Algorithm ................. 193
    5.6.1.2 Implementation of the Algorithm in LISP ................... 197
  5.6.2 Resolution-Method Theorem Prover .................................. 198
    5.6.2.1 The Basic Theorem Prover .................................. 198
    5.6.2.2 Optimizing the Theorem Prover ............................ 198
    5.6.2.3 Optional Additions ........................................ 199

6 Data Representation .................................................................. 201
  6.1 Representing Integers ...................................................... 201
    6.1.1 Unsigned Binary Numbers ....................................... 201
      6.1.1.1 Conversion of Binary to Decimal ......................... 202
      6.1.1.2 Conversion of Decimal to Binary ....................... 202
      6.1.1.3 Addition of Unsigned Binary Numbers ................. 204
    6.1.2 Signed Binary Numbers .......................................... 204
      6.1.2.1 Addition and Subtraction of Signed Binary Numbers ... 207
      6.1.2.2 Shifting Signed Binary Numbers ....................... 208
      6.1.2.3 Hexadecimal Notation ................................... 209
  6.2 Representing Characters .................................................. 210
  6.3 Representing Rational Numbers ........................................ 211
  6.4 Representing Programs .................................................... 212
  6.5 Memory Organization ....................................................... 213
    6.5.1 Units of Memory .................................................. 214
      6.5.1.1 Historical Perspective .................................. 214
    6.5.2 Addresses and Pointers ......................................... 214
  6.6 Exercises ........................................................................ 215
    6.6.1 ........................................................................ 215
    6.6.2 ........................................................................ 215
    6.6.3 ........................................................................ 215

7 A MIPS Tutorial ...................................................................... 217
  7.1 What is Assembly Language? ............................................ 217
  7.2 Getting Started: add.asm .................................................. 218
    7.2.1 Commenting .......................................................... 218
    7.2.2 Finding the Right Instructions .................................. 219
    7.2.3 Completing the Program ........................................... 220
      7.2.3.1 Labels and main ............................................. 220
## CONTENTS

7.2.3.2 Syscalls ................................................. 221
7.3 Using SPIM .................................................. 222
7.4 Using syscall: add2.asm .................................... 223
  7.4.1 Reading and Printing Integers .......................... 224
7.5 Strings: the hello Program .................................. 225
7.6 Conditional Execution: the larger Program .................. 227
7.7 Looping: the multiples Program .............................. 230
7.8 Loads: the palindrome.asm Program ......................... 231
7.9 The atoi Program ............................................ 234
  7.9.1 atoi-1 .................................................... 234
  7.9.2 atoi-2 .................................................... 236
  7.9.3 atoi-3 .................................................... 236
  7.9.4 atoi-4 .................................................... 237
7.10 Function Environments and Linkage ......................... 238
  7.10.1 Computing Fibonacci Numbers .......................... 240
    7.10.1.1 Using Saved Registers: fib-s.asm .................. 240
    7.10.1.2 Using Temporary Registers: fib-t.asm .............. 241
    7.10.1.3 Optimization: fib-o.asm ............................ 243
7.11 Structures and sbrk: the treesort Program ................. 245
  7.11.1 Representing Structures ................................ 245
  7.11.2 The sbrk syscall ........................................ 246
7.12 Exercises ................................................. 247
  7.12.1 .......................................................... 247
  7.12.2 .......................................................... 247
  7.12.3 .......................................................... 247
  7.12.4 .......................................................... 247
  7.12.5 .......................................................... 248
  7.12.6 .......................................................... 248
  7.12.7 .......................................................... 248
  7.12.8 .......................................................... 248

8 The MIPS R2000 Instruction Set .................................. 249
  8.1 A Brief History of RISC .................................... 249
  8.2 MIPS Instruction Set Overview .............................. 250
  8.3 The MIPS Register Set ...................................... 250
  8.4 The MIPS Instruction Set .................................... 251
    8.4.1 Arithmetic Instructions ............................... 252
    8.4.2 Comparison Instructions ............................... 253
# CONTENTS

8.4.3 Branch and Jump Instructions ........................................ 253
  8.4.3.1 Branch ......................................................... 253
  8.4.3.2 Jump ......................................................... 254

8.4.4 Load, Store, and Data Movement ....................................... 254
  8.4.4.1 Load ........................................................ 254
  8.4.4.2 Store ........................................................ 255
  8.4.4.3 Data Movement .............................................. 256

8.4.5 Exception Handling ..................................................... 256

8.5 The SPIM Assembler ....................................................... 257
  8.5.1 Segment and Linker Directives ................................... 257
  8.5.2 Data Directives ................................................ 258

8.6 The SPIM Environment ................................................... 258
  8.6.1 SPIM syscalls .................................................. 258

8.7 The Native MIPS Instruction Set ......................................... 258

8.8 Exercises ............................................................... 260
  8.8.1 ................................................................. 260

9 MIPS Assembly Code Examples .............................................. 261
  9.1 add2.asm ............................................................ 262
  9.2 hello.asm .......................................................... 263
  9.3 multiples.asm ...................................................... 264
  9.4 palindrome.asm ..................................................... 266
  9.5 atoi-1.asm .......................................................... 268
  9.6 atoi-4.asm .......................................................... 270
  9.7 printf.asm .......................................................... 272
  9.8 fib-o.asm ........................................................... 276
  9.9 treesort.asm ......................................................... 278

10 C for Programmers .......................................................... 285
  10.1 A Binary Search Program Prototype ................................ 285
    10.1.1 Binary Search Detailed Design .............................. 286
    10.1.2 Binary Search Code ......................................... 290
    10.1.2.1 Binary Search Test ...................................... 293
  10.2 The Basics of C .................................................... 294
    10.2.1 Syntax ....................................................... 294
    10.2.2 Data .......................................................... 297
    10.2.3 Expressions .................................................. 301
    10.2.4 Functions .................................................... 302
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2.5 Statements</td>
<td>303</td>
</tr>
<tr>
<td>10.2.6 Preprocessing</td>
<td>304</td>
</tr>
<tr>
<td>10.3 Adding Input to the Binary Search Prototype</td>
<td>305</td>
</tr>
<tr>
<td>10.4 Using Pointers in the Binary Search Prototype</td>
<td>309</td>
</tr>
<tr>
<td>10.5 The Final Binary Search Program</td>
<td>313</td>
</tr>
<tr>
<td>10.6 The Heap Sort Program</td>
<td>328</td>
</tr>
<tr>
<td>10.6.1 Heap Sort Header Files</td>
<td>328</td>
</tr>
<tr>
<td>10.6.2 Heap Sort Function Detailed Designs</td>
<td>337</td>
</tr>
<tr>
<td>10.6.3 Heap Sort Program Makefile</td>
<td>350</td>
</tr>
<tr>
<td>11 C++</td>
<td>353</td>
</tr>
<tr>
<td>11.1 Introduction</td>
<td>353</td>
</tr>
<tr>
<td>11.1.1 What is C++?</td>
<td>353</td>
</tr>
<tr>
<td>11.1.2 Non-Object-Oriented Features of C++</td>
<td>353</td>
</tr>
<tr>
<td>11.1.3 Object Orientation in C++</td>
<td>354</td>
</tr>
<tr>
<td>11.1.4 Data Abstraction</td>
<td>355</td>
</tr>
<tr>
<td>11.1.5 Class Inheritance</td>
<td>357</td>
</tr>
<tr>
<td>11.2 Non-Object-Oriented Features of C++ that Conflict with ANSI C</td>
<td>360</td>
</tr>
<tr>
<td>11.2.1 Non-Object-Oriented Features of C++ that Extend ANSI C</td>
<td>361</td>
</tr>
<tr>
<td>11.2.1.1 Inclusion of C Library Headers in C++ Files</td>
<td>361</td>
</tr>
<tr>
<td>11.2.1.2 Definition of Aggregate Data Types</td>
<td>361</td>
</tr>
<tr>
<td>11.2.1.3 Naming Conflicts between Variables and Data Types</td>
<td>362</td>
</tr>
<tr>
<td>11.2.1.4 Use and Scope of Symbolic Constants</td>
<td>362</td>
</tr>
<tr>
<td>11.2.2 Non-Object-Oriented Features of C++ that Extend ANSI C</td>
<td>363</td>
</tr>
<tr>
<td>11.2.2.1 The // comment delimiter</td>
<td>363</td>
</tr>
<tr>
<td>11.2.2.2 Position of Variable Definitions</td>
<td>364</td>
</tr>
<tr>
<td>11.2.2.3 Memory Management: The new and delete operators</td>
<td>365</td>
</tr>
<tr>
<td>11.2.2.4 Function Inlining</td>
<td>367</td>
</tr>
<tr>
<td>11.2.2.5 Function Overloading</td>
<td>368</td>
</tr>
<tr>
<td>11.2.2.6 The Reference Data Type</td>
<td>369</td>
</tr>
<tr>
<td>11.2.2.7 Default Function Arguments</td>
<td>371</td>
</tr>
<tr>
<td>11.2.3 Input and Output with C++</td>
<td>372</td>
</tr>
<tr>
<td>11.2.3.1 The &gt;&gt; and &lt;&lt; Input and Output Operators</td>
<td>373</td>
</tr>
<tr>
<td>11.2.3.2 Operator Overloading with &lt;&lt; and &gt;&gt;</td>
<td>374</td>
</tr>
<tr>
<td>11.3 Object Oriented Features of C++</td>
<td>377</td>
</tr>
<tr>
<td>11.3.1 Encapsulation in the Real World</td>
<td>377</td>
</tr>
<tr>
<td>11.3.2 Encapsulation in C++ : Interface Specification</td>
<td>378</td>
</tr>
<tr>
<td>11.3.3 Encapsulation in C++ : Implementation</td>
<td>380</td>
</tr>
</tbody>
</table>
11.3.3.1 Encapsulation at Work: An Example .......................... 381
11.3.4 Telling Objects Apart ............................................. 381
11.3.5 Initialization of Objects: Constructors .......................... 383
  11.3.5.1 First Tries .................................................. 383
  11.3.5.2 A Better Approach: Constructors ......................... 383
  11.3.5.3 Overloading Constructors .................................. 385
11.3.6 Deinitialization of Objects: Destructors ......................... 386
11.3.7 Friends .......................................................... 386
11.3.8 Subclasses and Inheritance ...................................... 387
11.3.9 Special Member Functions ...................................... 390
  11.3.9.1 Dereferencing: operator-> ................................ 391
  11.3.9.2 Conversion Operators ...................................... 391

12 Lexical Analysis and Parsing ........................................ 393
  12.1 An Arithmetic Language .......................................... 393
  12.2 The Building Blocks of Language Analysis ...................... 395
  12.3 The Lexical Analyzer .............................................. 396
  12.4 The Parser ....................................................... 406
  12.5 Semantics ......................................................... 414
  12.6 Parsing Projects .................................................. 422
    12.6.1 Regular Expressions and Pattern-Matching ................ 422
      12.6.1.1 Strings and Regular Expressions ...................... 422
      12.6.1.2 Non-Deterministic Finite Automata ................... 425
      12.6.1.3 Representing Regular Expressions with NDFAs ...... 426
      12.6.1.4 Implementing NDFAs in C ............................. 428
      12.6.1.5 Simulating the Non-Deterministic Finite Automaton 429
      12.6.1.6 Optional Additions .................................... 432
    12.6.2 Extending the Arithmetic-Language Interpreter ............ 432
      12.6.2.1 A New and Improved Language ......................... 432
      12.6.2.2 Implementing a Parse Tree ............................ 434
      12.6.2.3 Modifying the Old Interpreter ......................... 438
        12.6.2.3.1 The Lexical Analyzer ............................ 438
        12.6.2.3.2 The Parser ........................................ 439
        12.6.2.3.3 Semantics ....................................... 440
        12.6.2.3.4 The Parse-Tree Processor ......................... 442
      12.6.2.4 De-Allocating Memory ................................. 443
      12.6.2.5 Optional Additions .................................... 444
    12.6.3 Follow-up to the Arithmetic Interpreter: Compilation ...... 445
| 12.6.3.1  | Introduction to Assembly Language | 445 |
| 12.6.3.1.1 | Registers | 445 |
| 12.6.3.1.2 | Instructions | 446 |
| 12.6.3.1.3 | Addressing Modes | 447 |
| 12.6.3.2 | Compiler Semantics | 450 |
| 12.6.3.3 | An Optimization | 455 |
| 12.6.3.4 | Comparisons and Branches in Assembly Language | 456 |
| 12.6.3.5 | Optional Additions | 458 |
This is the CS 51 course project book, which is the textbook for CS 51, including background material and exercises. The project book is intended to be supplemented by programming language textbooks and course handouts.

CS 51 is defined as a course that begins by teaching LISP, ends by writing a LISP interpreter in C++, and contains in between some great ideas of computer science expressed in either LISP or C++. CS 51 also includes a brief study of assembly language. The course emphasizes teaching students how to reduce algorithms to code, by giving exercises in coding small but non-trivial algorithms. Both the course and the project book are dynamic, changing from year to year.

The CS 51 project book is written in Latex and is available by anonymous ftp in the pub/cs51 directory on endor.das.harvard.edu. The full Latex project book contains the chapters that appear in this printing, a historical record of chapters from past years, and proposed chapters that have not been used yet. Each year the professor selects chapter-versions for a year-specific version of the project book.

The course and project book are be UNIX based, but the code is to be platform independent, so students may run examples and develop solutions on MACs or PCs.

The project book is written by volunteers who want to try their hand at writing a textbook. The current list of contributors is: Harry Lewis, Jason Abrevaya, Bob Walton, Sanjoy Dasgupta, Dan Ellard, Bolade Gbadegesin, Jon McAuliffe, Mark Immel.

CS 51 teaching fellows, present and past, and similarly qualified persons are invited to contribute chapters or sections to the Latex version of the project book for possible future use. Instructions for doing so are in the README file of the current Latex version. The Latex version also contains a more detailed record of the contributors to each section.
Chapter 1

The Structure of LISP

by Jason Abrevaya, Bob Walton, Harry Lewis, Sanjoy Dasgupta

1.1 Introduction to LISP

This section gives a brief introduction to LISP. It is intended to get the reader started, but is not a replacement for a good COMMONLISP textbook. The reader should read some COMMONLISP textbook simultaneously with this chapter.

1.1.1 S-Expressions in LISP

Lisp was originally developed, in part, to perform symbolic mathematical calculations, such as:

\[
\frac{d}{dx} \left( \frac{1}{x+1} \right)^2 = 2 \frac{1}{(x+1)(x+1)^2} - \frac{1}{(x+1)}
\]

Later in the course we will in fact write a LISP function, \texttt{deriv}, such that

\[
(\texttt{deriv '(** (/ 1 (+ x 1)) 2) 'x})
\]

\[
(* 2 (* (/ 1 (+ X 1)) (/ -1 (** (+ X 1) 2))))
\]

which performs the indicated calculation.

In order to perform such calculations LISP must have a way of representing symbolic mathematical expressions. To this end, LISP has a type of data, the LISP s-expression, or “symbolic expression”, which LISP uses to express data. The s-expression is very general, and can express any type of data that can be written in a mathematical notation, including program code. Therefore s-expressions are used for both data and code in LISP.
CHAPTER 1. THE STRUCTURE OF LISP

The two basic types of s-expression are atoms and lists. An atom can be either a symbol (e.g. a, how-how-brown-cow, cons) or a number (e.g. 3.14, -22, 13470246). A symbol is any sequence of non-blank characters which is not a number, and which does not include the characters ‘ ’ ~ ( ) : , ; ‘ \ # or |. A list, meanwhile, is any sequence of s-expressions (i.e., atoms or lists) enclosed by parentheses. Some examples of lists are:

(a b c)
(foo (a b c) 3)
(foo (foo (foo 1 2) (foo 3 4)) 5)
()

Each of the first three examples is a list of three s-expressions. The first list is simply a list of three atoms (all symbols); the second and third are both lists of a symbol, a list, and a number. In the third example, the list’s second s-expression is itself a list containing the foo symbol and two additional lists. The final example is a special one in that there are no s-expressions contained within the parentheses. This empty list is represented in LISP by the symbol nil, so that () and nil are two different ways of denoting the same thing. The empty list is the only LISP s-expression that is both an atom and a list.

We can express the general form of an s-expression using a simple recursive grammar:

```
S-EXPR ::= ATOM | LIST (1)
LIST ::= ( S-EXPR* ) (2)
```

In this notation:

::= means “is a,”
| means “or,”
* means “zero or more,”

While this grammar accurately describes the s-expressions that have been discussed up to this point, it turns out that the grammar is actually too simple to describe all possible s-expressions in LISP. That is, there is more to LISP than just lists and atoms. We’ll improve upon our grammar later when we discuss explicit dotted-pair forms, but for now it’s a useful exercise to go through the aforementioned examples to check that they satisfy the rules of the grammar. Looking at the third example, we can check it against the grammar as follows:

```
(foo (a b c) 3)  ->  (ATOM (ATOM ATOM ATOM) ATOM)
   ->  (ATOM (S-EXPR S-EXPR S-EXPR) ATOM)  by (1)
   ->  (ATOM LIST ATOM)  by (2)
   ->  (S-EXPR S-EXPR S-EXPR)  by (1)
   ->  LIST  by (2)
   ->  S-EXPR  by (1)
```
1.1. INTRODUCTION TO LISP

Above we mentioned a list of characters that could not be included in a symbol. Actually, these characters, except for | and \, can be included in a symbol, if the symbol is surrounded by | characters, which "quote" the symbol. Also, a character can be included in a symbol if it is preceded by a \, which "escapes" the character. Some examples:

-> (progn (princ '|This is a quote: ".|) (terpri))
This is a quote: ".
NIL

-> (progn (princ '|This is a backslash: \\\.|) (terpri))
This is a backslash: \\\\.
NIL

-> (progn (princ 'This\ is\ a\ vertical\ bar:\ \\.|) (terpri))
THIS IS A VERTICAL BAR: \\.
NIL

1.1.2 LISP Interpretation and Compilation

When a computer reads a program, it first stores the code in a form similar to what a person sees when the person reads the code. A program that reads the code in this form and executes that code is called an interpreter. A program that reads the code in this form and translates it to a different form that is directly understood by computer hardware is called a compiler.

Originally the LISP language was only executable by an interpreter, but later the language was refined so that it could also be compiled. The advantage of compilation is that the compiled code runs faster, and the disadvantage is that you have to wait until the compiler finishes before you can test. Also, to gain speed, error checking is often omitted from the compiled code, which makes debugging harder, though you can tell the compiler not to do this.

Programming in an interpreted language is generally faster than in a compiled language. The main reason is that debugging is faster if you do not have to wait for compilations and if you have error checks done for you.

A LISP interpreter can engage its user in a dialog in which the user types an s-expression into the interpreter, and the interpreter evaluates that s-expression and prints the resulting value s-expression. One of the best ways to learn interpreted languages like LISP is to read through some description of the language and try out each new feature as you read about it, using the interpreter. You may want to do this, using one of the COMMONLISP text books recommended by this course.

We will do a little of this here in order to introduce the language, the interpreter, and the compiler. The following is a sample interpreter dialog:
An s-expression that is just a number evaluates to itself.

An s-expression that is a list whose first element is a symbol naming a function, such as +, evaluates by calling the function with the other elements of the list as arguments. For example, (+ 4 66) above evaluates to 70.

The arguments may be s-expressions, and they are evaluated before they are used. For example:

(\(+ 4 (* 6 11)\))

Here the arguments to + are 4 and (* 6 11).

Symbols may be used to name variables, as in:

(setf x 66)

(setf x 66)

So far we have given only s-expressions that evaluate to numbers. Here are some s-expressions that evaluate to symbols and lists:
1.1. INTRODUCTION TO LISP

-> ' (this is a list) ; ' quotes literal data
(THIS IS A LIST)
-> ' hello
HELLO
-> (append ' (first list) ' (second list))
(FIRST LIST SECOND LIST)
-> () ; "()" denotes nil
NIL
-> (car ' (one two three four)) ; "car" means "first"
ONE
-> (cdr ' (one two three four)) ; "cdr" means "rest"
(TWO THREE FOUR)
-> (null ' (one two three four)) ; "null" tests for empty lists
NIL
-> (null '())
; nil means "false", T means "true"
T
-> (null nil)
; nil is also the same as "()"
T

To keep a list from being evaluated, you quote it: i.e., you put a single quote in front of it. Similarly to keep a symbol from being evaluated, you quote it (when an unquoted symbol is evaluated, it should name a variable, and the value of the variable is the result).

Usually letters read into the interpreter are changed to capitals immediately, stored as capitals inside memory, and printed as capitals.

The append function concatenates lists.

The empty list () is represented in LISP by the symbol nil, and the empty list prints as nil, and not as (). On input you can use either () or nil to mean exactly the same thing.

A semi-colon ; starts a comment, which continues till the end of its line.

COMMONLISP has two functions, car and first, that do exactly the same thing: they return the first element of a list. Similarly COMMONLISP has two functions, cdr and rest, that do exactly the same thing: they return the rest of the list excluding the first element. The names car and cdr are historical, but they are also the names all LISP programmers regularly use for these functions, and so they are the names we will use.

The null function just returns true if its argument is an empty list, and false otherwise. In LISP false is represented by the symbol nil, and true is represented by any non-nil value, with the symbol t being used when nothing else is handy.

The interpreter is good at detecting errors, such as trying to get the first element of something that is not a list. For example, consider the following function that returns true if its argument is a
list that has at most one element:

```lisp
-> (defun short-list (lst) (null (cdr lst)))
SHORT-LIST
-> (short-list '(fee fi fo fum))
NIL
-> (short-list '(fee))
T
-> (short-list 'fee)
Error: FEE is not of type LIST
```

It is possible to compile functions to make them run faster. However, compiled code often does not have the same amount of error checking as interpreted code. For example:

```lisp
-> (compile 'short-list) ; compiles previously defined function
SHORT-LIST
-> (short-list '(fee))
T
-> (short-list 'fee)
NIL
```

Here NIL is not the right answer: there is no right answer, as ’fee does not evaluate to a list.

### 1.1.3 Introduction by Analogy with C (or Pascal)

Some of LISP is just like C (or Pascal) except for changes of syntax and names.

First, LISP function calls are written without commas separating arguments, and with the first left parentheses placed before the function name instead of after it. Thus the C expression `max(y,5)` is written in LISP as `(max y 5)`. Examples:

```lisp
-> (max 8 3)
8
-> (atan 0 -1) ; Arctangent of first-argument/second-argument.
3.141593
```

Second, LISP has no infix or prefix operators, but uses function call notation for everything. Thus the C expression `x + 5 * y` becomes the LISP expression `(+ x (* 5 y))`. Example:

```lisp
-> (+ 4 (* 3 7))
25
```

Third, LISP uses somewhat different names for some things: see Figure 1.1. Examples:
### 1.1. INTRODUCTION TO LISP

<table>
<thead>
<tr>
<th>C Name</th>
<th>Pascal Name</th>
<th>LISP Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>=</td>
<td>:=</td>
</tr>
<tr>
<td>numeric comparison</td>
<td>==</td>
<td>=</td>
</tr>
<tr>
<td>pointer comparison</td>
<td>==</td>
<td>=</td>
</tr>
<tr>
<td>numeric comparison</td>
<td>!=</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>pointer comparison</td>
<td>!=</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>floating point divide</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>rem integer divide</td>
<td>/</td>
<td>† no equivalent</td>
</tr>
<tr>
<td>remainder</td>
<td>%</td>
<td>† no equivalent</td>
</tr>
<tr>
<td>mod integer divide</td>
<td>/</td>
<td>† div</td>
</tr>
<tr>
<td>modulo</td>
<td>%</td>
<td>† mod</td>
</tr>
<tr>
<td>bitwise inclusive or of integers</td>
<td></td>
<td>no equivalent †</td>
</tr>
<tr>
<td>bitwise and of integers</td>
<td>&amp;</td>
<td>no equivalent †</td>
</tr>
<tr>
<td>bitwise complement of integer</td>
<td>~</td>
<td>no equivalent †</td>
</tr>
<tr>
<td>bitwise exclusive or of integers</td>
<td>^</td>
<td>no equivalent</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td>logical not</td>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td>if-then-else operators</td>
<td>? :</td>
<td>no equivalent</td>
</tr>
</tbody>
</table>

†The C integer / and % operators perform either remainder or modulo style division, depending upon whichever is more efficient on the target hardware.

‡Pascal has no bitwise integer operations, but does have set operations with operators +, *, and – for set union, intersection, and difference.

![Figure 1.1: Lisp-C-Pascal Name Correspondence](image-url)
CHAPTER 1. THE STRUCTURE OF LISP

-> (setf x 5)
5
-> x
5
-> (truncate 38 x)
7
-> (rem 38 x)
3
-> (/ 38.0 5.0)
7.6
-> (logior 3 x)
7
-> (logand 3 x)
1
-> (logxor 3 x)
6
-> (if (< 3 2) 33 22)
22

However, LISP uses the same name as C for the following operators when they are applied to numbers:

```
+  -  *  >  <  =>  <=
```

Fourth, in LISP an expression is true (for the purposes of if) if and only if it is non-nil, just as in C an expression is true if and only if it is non-zero. And if no other non-nil value is handy, LISP uses the symbol \texttt{t} for true, just as if no other non-zero value is handy, C uses the number 1 for true. Examples:

-> (= 9 7)
NIL
-> (= 9 9)
T
-> (eq 'x 'y)
NIL
-> (eq 'x 'x)
T
1.1. INTRODUCTION TO LISP

-> (or nil t nil)
T
-> (and nil t nil)
NIL
-> (not nil)
T

Fifth, arrays in LISP can be created by the function make-array, just as arrays in C can be created by the function malloc. The make-array function takes a single argument that is a list of the dimension sizes, e.g. `(3 4). Array elements in LISP can be referenced by using subscripts: the C expression `x[5][i] becomes `(aref x 5 i) in LISP. However, you cannot create arrays by declaration in LISP, nor can you create pointers to elements of an array. Like C, the first row or column corresponds to the subscript 0. Examples:

-> (setf x (make-array '(3 4))) ; 3 rows and 4 columns.
#2A((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL))
-> (setf (aref x 0 2) 55) ; Set element in 1st row, 3rd column.
55
-> x
#2A((NIL NIL 55 NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL))
-> (aref x 0 2) ; Read element in 1st row, 3rd column.
55

LISP is also rather different from C in some ways.

First, there are differences between the concept of a value in LISP and and a value in C. In C, each variable has a type that determines the values that may be stored in the variable, and in particular the size of these values. In LISP, any variable can store any value, regardless of the type or size of the value (e.g. an arbitrarily large integer or arbitrarily long list may be stored). How LISP does this is the subject of a separate section: see Internal Representation of LISP Values. Examples:

-> (setf x 99)
99
-> (+ 112233445566778800 x)
112233445566778899
-> (setf x '(this is a list))
(TTHIS IS A LIST)
-> (cdr x) ; cdr means "rest of list after first element"
(IS A LIST)

Second, LISP uses symbols for both names and character strings. An example of the character string use is:
CHAPTER 1. THE STRUCTURE OF LISP

-> (progn (princ '|All throbsy were the solvophiles!|) (terpri))
All throbsy were the solvophiles!
NIL

Here the symbol name is quoted with | characters so that it can contain characters such as lower case letters, the single space character, and the apostrophe. The prin function prints symbol names without |’s, and the terpri function prints an end of line. We also note that COMMON-LISP has a separate string data type that we do not use in this course.

Third, in LISP data is often represented by s-expressions and entered into programs by quoting s-expressions:

-> '((george 55) (bill 33) (jane 101))
((GEORGE 55) (BILL 33) (JANE 101))
-> 'Hello
HELLO

Fourth, pieces of LISP programs can be used as data, and LISP data in the proper format can be used as programs. The following is an example (which may be too complex for the reader to understand precisely at this time):

-> (setf arglist ' (x))
(X)
-> (setf expression ' (* 2 x))
(* 2 X)
-> (list 'lambda arglist expression) ; the list function makes a list of its arguments
(LAMBDA (X) (* 2 X))
-> (setf my-fun (compile nil (list 'lambda arglist expression)))
#<compiled-function COMPLIER::ANONYMOUS-LAMBDA>
-> (funcall my-fun 33) ; funcall calls a function
66

The expression calling compile takes a list of argument names, arglist, and an expression, and produces a compiled function that evaluates the expression given values for the named arguments.

Fifth, LISP statements and declarations, like defun for defining functions, cond for conditional execution, and do for iteration, have the syntax of function calls, though their arguments are treated quite differently from function call arguments, and may have special syntactic structure. An advantage is that in LISP new statements and declarations may be defined easily as macros. The following are some examples (which may be too complex for the reader to understand very well at this time):
1.2. Internal Representation of LISP Values

One of the mysteries of LISP is how any variable can hold any value. After all, some lists are long, and some are short; and LISP integers can be any length, even 1,000 digits.

The secret is that LISP variables do not store values: they store pointers to values. Since all pointers are the same size, all variables can be that size, and any variable can store any value.

As long as the values pointed at are not changed, it is perfectly OK to confuse pointers somewhat with the values they point at. The LISP statement

\[
\text{(setq } x \ y)\]

copies the LISP value of the variable \( y \) to the variable \( x \). But what really happens is the pointer stored in \( y \) is copied to \( x \). As long as the value pointed at is not changed, this is just as good as copying the value. But if the value pointed at were changed, both the LISP value of \( x \) and the LISP value of \( y \) would appear to change simultaneously and identically.

Now suppose we write a program that sets the variable \( x \) to 0 and then executes

\[
\text{(setq } x \ (+ \ x \ 1))\]

10,000,000 times. What happens to memory as this program executes?

First, some memory is allocated to store 0, and a pointer to this is stored in \( x \), so memory looks like:
Then some memory is allocated to store 1, and a pointer to this is stored in x:

```
  x
  | 0
```

Then 2 is allocated, then 3, and so forth, till 10,000,000:

```
  x
  | 10,000,000
```

Won’t we run out of memory?

What saves us is that after we store a pointer to 1,000 in x, we no longer need the value 999. Nor do we need 998, 997, 996, …. Nothing is pointing at any of these values. You cannot reach these values by starting from variables and following pointers. They are unreachable. They are garbage.

And LISP has a garbage collector, upon which it very much depends. When LISP finds that it is running out of memory, it stops and calls the garbage collector, which finds all the garbage, and frees the memory occupied by that garbage, so it can be reused to allocate new values. Then LISP continues.

We will discuss garbage collectors in much more detail near the end of this course. For now, the important thing is to understand that the LISP way of handling values requires a garbage collector and introduces important distinctions between changing a variable and changing a value.

Changing variables merely involves changing the pointers contained in the variables, and causes no problems.

The safe way to change a value, like a three element list whose first element you want to change, is to make a copy with the alteration, and store a pointer to the altered copy in the variable of your choice. This way any other variables pointing at the original value will not appear to change unexpectedly, because the original value is not changed.

The unsafe way to change a value is to edit the original value directly, as if the value itself were variable. Then all variables that point at the value would change, and unless you expect this behavior, your program might develop bugs. This unsafe way is called destructive, because the original value is destroyed.
LISP functions that change values destructively are called *destructive functions*. We will avoid these as much as possible, and not talk about them until later.

Is `setf` of a variable a destructive function? We will answer no, it is not, because it changes something that is assumed to be changeable. For example, variable values are not printed when one prints a symbol; only the symbol name is printed. Changing a variable does not change the apparent, printed value of other variables. However, when `setf` is used to change things like the first element of a list, as in `(setf (car x) y)`, then it is destructive.

The LISP values pointed at by the variables may be numbers, symbols, lists, or arrays (or other things not covered in this course). All these values are multicomponent structures, in the sense of C structures or PASCAL records. The components of these structures are summarized in Figure 1.2.

(There may be other components of values which we do not discuss: for example, implementation dependent components used for garbage collection.)

In the following subsections we will discuss in detail how the values just summarized are stored. We will then investigate how storage of LISP values effects the ways one tests for equality of these values.

Students in this course are advised that they do not have to master all the details in this section at the very beginning of the course. Some of these details will be emphasized in lecture at the beginning, but others will be left to the end where the LISP interpreter is studied in some detail.

### 1.2.1 Internal Representation of LISP Numbers

As explained above, LISP numbers are values stored in memory and pointed at by LISP variables. LISP never changes the number values stored in memory: if it wants to store a new LISP number in a variable, it allocates new memory to hold the new value. This is unlike LISP lists, whose values can be changed. Another way of saying this is that LISP has no destructive functions that change number values.

What is in a number value?

The first component of a number value tells the type of the value (see Figure 1.2). This is necessary because there are several types of numbers — integers, ratios, and floating point — and arithmetic operators like `+` and `*` need to know what type of number they are operating on. The type is first because operators must know the type in order to know how the rest of the number value is formatted in memory.

The `type-of` function can be used to discover the type of a number value (or, for that matter, any LISP value). Here are some examples which we will explain better below:
Figure 1.2: Summary of LISP Value Components

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Type</th>
<th>Value</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fixnum</strong></td>
<td></td>
<td><strong>bignum</strong></td>
<td></td>
<td><strong>ratio</strong></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
<td>type</td>
<td>sign</td>
<td>type</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td>sign</td>
<td>numerator</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>number</td>
<td>denominator</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>of digits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>digits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>short-float</strong></td>
<td></td>
<td><strong>long-float</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td>value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>symbol</strong></td>
<td></td>
<td><strong>cons</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
<td>car</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>array</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.2. INTERNAL REPRESENTATION OF LISP VALUES

-> (type-of 55)
FIXNUM

-> (type-of 555555555555)
BIGNUM

-> (type-of 5/8)
RATIO

-> (type-of 5.5)
LONG-FLOAT

-> (type-of 5.5s0)
SHORT-FLOAT

There are two integer types: fixnum and bignum. fixnum integers are small and efficient; bignum integers are large and less efficient.

If a number is a small integer, a fixnum, the second component (in a typical LISP implementation) is the value of the integer stored in 32 bits. Other efficiency tricks are used for storing fixnum integers, but we will refrain from mentioning these till the end of this section.

If the number is a large integer, a bignum, the second component is the sign of the integer, the third component is the number of digits in the integer, and the fourth component is these digits. (These digits, by the way, are not decimal, and usually not even exactly binary. In a typical LISP implementation they are digits to the base $2^{32}$, so each digit fits exactly into one 32 bit word of the computer.)

LISP also has rational numbers, of type ratio, e.g.:

-> (+ 5/8 1/4)

7/8

A rational number is a pair of integers, one for the numerator, and one for the denominator. The second and third components of a rational number value are pointers to these integers (in a typical LISP implementation). (These integers are typically required to obey some rules, like the denominator is positive and not zero, and the numerator and denominator cannot have any common divisor.)

A floating point number (in a typical LISP implementation) just has a second component which is either a 32-bit or 64-bit floating point number coded according to the IEEE standard for coding floating point numbers. (The IEEE standard tells how a sign, exponent, and mantissa are to be encoded in 32 or 64 bits, but this is not a concern of this course.) The length of the floating point number, 32 or 64 bits, is determined by the type of the number, which is either short-float or long-float.

\[^1\)fixnum is an historical name that is a bit of an anachronism: in early LISP the number types were fixnum for integers and flonum for floating point, but while the name flonum has disappeared, fixnum remains with us.]
In this course we are not very interested in the rational or floating point numbers, or their details. We are more interested in the integers.

Doesn’t it seem rather inefficient to allocate a value to memory every time you need a new value? Isn’t it particularly annoying to have to do this for every small integer value a program generates? Won’t the programs run slowly if we do this? Yes, yes, and yes, of course.

So what is the solution? The basic idea is to segregate out sufficiently small integers, and impose a linear relationship between the integer represented and the pointer to the value of that integer. In other words, we impose a mathematical equation of the form:

$$\text{pointer} = M \times \text{integer} + N$$

where $M$ and $N$ are some constants (e.g. $M = 8$ bytes and $N = \text{some address}$). Then, when we are asked to allocate an integer value to memory, we check whether the integer is a small integer, and if yes, compute the pointer to its value, and forget about actually allocating memory.

The actual small integer values must be prestored in some way. One way to do this is to just do it. Another tricker way is not to store anything at all, ever, where the small integer pointers point, and every time you have to read a value using a pointer, check first to see if its a small integer pointer, and if yes, compute the value from the pointer instead of trying to read it from memory.

For the user of LISP none of this matters too much, except in the following ways. First, he can expect computations with small integers to be much faster than computations with other numbers, because no memory need be allocated. Second, he will notice differences in how the $eq$ operator compares small integers and larger integers. This operator compares the pointers to the values of the numbers, and does not compare the values themselves. For small integers the pointers are equal if and only if the integers are equal, because of the above equation, and $eq$ does the right thing. For large integers, the pointers may not be equal even if the integers are equal, and $eq$ does the wrong thing. More on $eq$ in a section below.

On many systems, all fixnum integers are stored using the above method. On some systems, only a smaller range of fixnum values are so stored.

### 1.2.2 Internal Representation of LISP Symbols

A LISP symbol is like a character string with some special properties.

The first component of a LISP symbol value is its type, which is `symbol`, as in:

```lisp
=> (type-of 'x)
SYMBOL
```

Another component of a LISP symbol is its name, which is just the character string that names the symbol. (Actually, in a typical LISP implementation, the name component is a pointer to the character string, and not the string itself).
Notice we did not say the name component is the second component. It isn’t, because of trickery we will describe below.

The first special property of a symbol is that no two different symbol values in memory have the same name. That is, when LISP reads the name of a symbol, it does not simply allocate a new symbol value to memory and start using a pointer to that value. Rather, it looks through all of memory to see if any symbol values with the same name are already there, and if yes, uses the pointer to the pre-existing value (there can be at most one), without allocating any new value. Only if it finds no pre-existing symbol value with the same name does it allocate a new symbol value.

Therefore, given two pointers at symbol values, we know the names of the values are equal if and only the pointers are equal. This is part of the secret of LISP’s speed as an interpretive language: it does not have to compare character strings, but only pointers, in order to handle names.

(It might be useful to note here that when LISP reads a symbol name, it first changes all lower case letters to upper case, before it does anything else, so the letters in a symbol name are generally stored as upper case. You can prevent this case change by surrounding the name with vertical bars, as in:

```
->’|This is a sentence.|
|This is a sentence.|
```

Here the vertical bars are not part of the name, but are sometimes printed on output to indicate the name contains lowercase letters and other special characters such as spaces.)

The other special property of symbols is that they name things. In particular, they name variables, functions, and property lists.

One symbol can name several variables. For example:

```
(set ’x 1) ; Set top level, or global, variable X.
(let ((x 2)) ; Create local variable X.
  (let ((x 3)) ; Create another local variable X.
    (print x) ; Prints 3, value of innermost
    ; variable named X.
    (print (eval ’x)) ; Prints 1, value of global variable X.
  )
)
```

A variable in LISP is just a place to store a pointer. Any number of such places can be named by the same symbol, as indicated above. The global variable named by a symbol is a component of the symbol value (in a typical LISP implementation). If this global variable has never been set, it has no value, and is considered to be unbound. To indicate this, unbound variables are given a
special implementation dependent value which cannot be a pointer to a LISP value, such as zero (which in C equals NULL).

Just to make life more interesting, a symbol can be declared to be a *special symbol* by mentioning it in a `defvar` statement. Special symbols are optimal for use in naming global variables, while non-special symbols are optimal for naming local variables. A symbol has a component, the symbol’s *special flag*, that is one bit which is turned on if the symbol is special, and off otherwise. Special symbols cannot in fact name local variables; they can only name global variables. We will not discuss this further here: see the section on *LISP Variables*.

The term “variable” can be a misnomer: often after a variable is given an initial value, it is never changed. So it is really a “constant”.

To escape from this terminological dilemma, the word “binding” is sometimes used. One says that a value is *bound* to a symbol, and such a binding is what we have called a variable. It follows that a symbol can have many values bound to it, one for each local variable named by the symbol, and one for the global variable named by the symbol. These bindings are called *variable bindings* to distinguish them from the function bindings that we are about to describe.

One symbol can name several functions. For example:

```lisp
(defun foo (x) (+ x 1)) ; Define top level, or global, function FOO.
(flet ((foo (x) (+ x 2))) ; Create local function FOO.
  (print (foo 10)) ; Prints 13, using innermost FOO.
  (print (funcall 'foo 10)) ; Prints 11, using global function FOO.
  (print (funcall #'foo 10)) ; Prints 13, using innermost FOO.
)
```

The situation is analogous to variables. A function definition is a place to store a pointer to a function. The global function definition is a component of the symbol value. The global function definition can be changed like a variable using code such as:

```lisp
(setf (symbol-function 'foo) #'(lambda (x) (+ 100 x)))
```

but local function definitions, introduced by `flet`, cannot be changed. Also, there is no function definition analogue to special symbols for variable definitions.

Sometimes the function definitions of a symbol are called the *function bindings* of the symbol.

We will briefly mention the kinds of function definitions that are possible. Each function definition of a symbol can be either an ordinary function or a macro. To make the global definition a macro, `defmacro` is used in place of `defun`. To make a local definition a macro, `macrolet` is used in place of `flet`. 
Recursive functions cannot be defined with \texttt{flet}, but may be defined with the \texttt{labels} special form, which is just like \texttt{flet} but permits functions to reference themselves recursively.

A few symbols, e.g. \texttt{if} and \texttt{let}, name \textit{special forms} that look like function or macro calls but are neither. No function definitions should be given for special form symbols. Both the compiler and interpreter contain so much knowledge about how to execute special forms that it is impractical to change the definition of a special form.

If the value of a function definition is a function, it may be called via \texttt{funcall} and \texttt{apply} and retrieved via \texttt{symbol-function}. If the value is a macro or the symbol is a special form, the value is implementation dependent, and should not be called via \texttt{funcall} or \texttt{apply} or retrieved via \texttt{symbol-function}.

More details concerning symbol function definitions are given in the section \textit{LISP Functions, Macros, and Special Forms}.

A symbol also names one and only one property list, which is a component of the symbol value. The global variable named by a symbol, the global function definition named by the symbol, and the property list named by the symbol are all components of the symbol value. You can read these components of the symbol \texttt{x} with the expressions:

\begin{verbatim}
(symbol-value 'x)
(symbol-function 'x)
(symbol-plist 'x)
\end{verbatim}

and write them with expressions such as:

\begin{verbatim}
(setf (symbol-value 'x) 55)
(setf (symbol-function 'x) #'(lambda (x) (+ x 55)))
(setf (symbol-plist 'x) '(height 55 weight 66))
\end{verbatim}

All three of these components are variables that store pointers to values (though the global function component is normally never changed once set). In some ways, however, these components are not considered to be part of the symbol’s value. They are not printed when the symbol is printed, and functions that change them are not considered to be destructive.

Above we mentioned trickery involving the ordering of the components of a symbol value. The problem is this: \texttt{nil} is a symbol, but it is also the empty list, and because it is we want to make the functions \texttt{car} and \texttt{cdr} return \texttt{nil} when applied to \texttt{nil}. What \texttt{car} does (in a typical LISP implementation) is return the second component of a list value (otherwise known as a dotted-pair value, see below). What \texttt{cdr} does is return the third component of a list value. (The first component of the list value, in case you were wondering, is the type). So — we make the second component of a symbol be the \texttt{symbol-value} component, and the third component be the \texttt{symbol-function} component. Therefore, if \texttt{car} is applied to a symbol its the same as \texttt{symbol-value}, and if \texttt{cdr} is applied to a symbol its the same as \texttt{symbol-function}. Then we insist that the global value of \texttt{nil} be \texttt{nil}, and the global function definition of \texttt{nil} also be
nil. Efficiency through trickery!

This is a good place to comment on whether such trickery is a good thing. First, this trickery is expensive: it takes time to explain, and time to understand, and if not understood, can cause implementors of LISP to make mistakes (e.g., one might forget to keep the symbol components in the right order). But it has a benefit: car and cdr run faster because they do not have to run a special check for nil every time they execute, and this in fact makes these heavily used functions several times faster in typical compiled LISP. So the trickery is a good thing if the value of the trickery exceeds its cost.

It’s moderately rare for the value of trickery to exceed its cost, so large programs (like LISP interpreters) usually have only a very few pieces of trickery.

1.2.3 Internal Representation of “Ordinary” Lists

A non-empty list value contains three components: the type, the car, and the cdr. These last two components correspond to the car and cdr functions in LISP, where car returns the first expression in a list and cdr returns the tail of a list (i.e., the list that remains when the first expression is removed)\(^2\). For example:

```
-> (type-of '(a b c))
CONS
-> (car '(a b c))
A
-> (cdr '(a b c))
(B C)
```

Here the type is cons, which means a cons cell, or value formed by the cons function. A cons cell is also called a dotted-pair for reasons which will become apparent in the next section.

Recall that the empty list, which is entirely different from non-empty lists, is represented by the symbol nil, and may be input as either nil or (), but will always be output as NIL. Note that:

\(^2\)COMMONLISP has the functions first and rest that have identical behavior to car and cdr and more reasonable names. It also has second, third, and fourth corresponding to cadr, caddr, and cadddr, and fifth,..., ninth that have no corresponding cad...dr. But there is no better-name version of cddr, cdar, etc.
1.2. INTERNAL REPRESENTATION OF LISP VALUES

-> (type-of '())
SYMBOL
-> (eq '() nil)
T
-> (car nil)
NIL
-> (cdr nil)
NIL

Here the car and cdr function are not really accessing car and cdr components of the symbol nil, because, being a symbol, nil has no such components. The car and cdr functions are treating the nil symbol as a special case. It is an error (often undetected in compiled code) to take the car or cdr of any symbol other than nil.

The car and cdr components store pointers to values, just like variables. However, unlike variables, the car and cdr components are presumed to be unchanging once they are set, because changing them changes the list value represented by the cons cell. Functions that do change the car or cdr components are called destructive functions, and should be used only with extreme care.

To diagram the internal representation of a list, we use dotted-pair box notation. For instance, the list (a b c) would be represented by a pointer to the following structure:

```
      —____—
      |     |
      |     |
      |     |
      ————
       |
       |
       |
     a     b     c     
     /     /     /     |
   ———— ———— ———— ————
     |     |     |     |
     |     |     |     |
     |     |     |     |
     ———— ———— ———— ————
       |
       |
       |
     nil
```

This notation is often simplified by replacing pointers to nil by slashes:

```
      —____—
      |     |
      |     |
      |     |
      ————
       |
       |
       |
     a     b     c
     /     /     /     |
   ———— ———— ———— ————
     |     |     |     |
     |     |     |     |
     |     |     |     |
     ———— ———— ———— ————
```

Each of the list values in this representation is known as a dotted-pair cell, with the first pointer pointing to the car of the dotted-pair and the second pointer pointing to the cdr of the dotted-pair:
With this in mind, we can fully describe the components of the dotted-pair box notation of (a b c) by labeling its pointers appropriately:

The list (a b c) is fairly simple in the sense that all of its atoms are on the top-level. However, dotted-pair box notation is not confined to representing lists having only top-level atoms. In fact, in more complicated cases, the notation is quite descriptive since all top-level atoms can be pictured in the first level of dotted-pair cells, second-level atoms in the second level of cells, and so on. Consider the diagram for (foo (a b c) 3):

Notice that the first pointer of the second dotted-pair cell points to the list (a b c). This indicates that (a b c) is the car of the cdr of (foo (a b c) 3), as expected.

1.2.4 Internal Representation of Dotted-Pair Forms

In the diagrams above, each of the cdr pointers points to a list (remembering, of course, that nil is considered a list). It is possible, however, to have cdr pointers pointing to non-nil atoms:
At first glance, the structure on the left looks very much like the representation of the list \((a \ b)\). However, the cdr pointer of the structure points to the atom \(b\), indicating that the cdr of the expression is not the list \((b)\). This expression is actually a different sort of creature than the lists which we’ve encountered until now. We use the notation \((a \ . \ b)\) to represent it, with the dot indicating that the cdr of \((a \ . \ b)\) is the single expression which follows the dot (the atom \(b\)) rather than the list of that expression (the list \((b)\)).

Using this same notation, the structure on the right is represented in LISP as \(((a \ . \ b) \ . \ (c \ . \ d))\).

If we look back in the previous section, we find that the list \((a \ b \ c)\) can be represented in dotted-pair notation as \((a \ . \ (b \ . \ (c \ . \ nil)))\). In fact, you can type this into LISP:

```
-> '(a . (b . (c . nil)))
(A B C)
```

When LISP reads an expression, it must produce dotted-pair cells. If it reads dotted-pair notation, it is clear how to do this. If it reads something else, it must have rules for converting it to dotted-pair notation.

Similarly, when LISP prints dotted-pair cells, it is clear how to print them in dotted-pair notation, but rules are needed if something else is to be printed instead.

The printer rules are easier to state, so we shall state them first. There are two rules. The first says that if the cdr of a dotted pair is a cons cell, omit the dot in front of it and also the parentheses around it. Thus

\[
(a \ . \ (b \ . \ (c \ . \ nil))) \Rightarrow (a \ b \ . \ (c \ . \ nil)) \Rightarrow (a \ b \ c \ . \ nil)
\]

by applying this rule twice. The second rule says that if the cdr is \texttt{nil}, omit it and the dot before it. Thus

\[
(a \ b \ c \ . \ nil) \Rightarrow (a \ b \ c)
\]

by this second rule. LISP applies these two rules when printing values so as to print representations with as few dots as possible.

The input rules can be given best by giving a grammar for LISP \textit{s-expressions} that is more complete than the grammar given in a previous section:

\[
\begin{align*}
\text{S-EXPR} & ::= \text{ATOM} \mid () \mid (\text{S-EXPR} \ \text{REST-OF-S-EXPR}) \quad (1) \\
\text{REST-OF-S-EXPR} & ::= ) \mid . \ \text{S-EXPR} \mid \text{S-EXPR} \ \text{REST-OF-S-EXPR} \quad (2)
\end{align*}
\]

Note that \((())\) is allowed as input, and represents \texttt{nil}, but this is never output.

The s-expression \(((a \ b) \ . \ c)\) fits the grammar as follows:
We can write a similar grammar for s-expressions as they are printed:

\[
\begin{align*}
S-EXPR & ::= A T O M \mid ( S-EXPR \ REST-OF-S-EXPR \ (1^*) \\
\text{REST-OF-S-EXPR} & ::= ) \mid . \ \text{N O N}-\text{NIL}-A T O M \mid (2^*) \\
& \mid \ S-EXPR \ \text{REST-OF-S-EXPR}
\end{align*}
\]

Note that () is never printed, and only a non-nil atom may appear after a dot.

Notice that these grammars concern themselves only with the syntactic correctness of an s-expression. There is no way to tell from the grammars whether a given s-expression is “meaningful” (e.g., in the sense that an interpreter would be able to evaluate it and return an appropriate value). In addition, the grammars which we have developed here are by no means intended to be a complete description of the LISP language. Most LISP systems require a more complex grammar to represent their languages, due to the existence of other objects, such as strings, arrays, etc.

### 1.2.5 Internal Representation of LISP Arrays

LISP arrays are just like cons cells, but with two differences.

First, instead of just two components, the car and the cdr, LISP arrays can have any number of elements, as long as they are organized in a rectangular array with some number of dimensions and some dimension sizes. Note that if an array dimension has size \( S \), then the subscripts for that dimension range from 0 through \( S - 1 \), or in other words, all subscripting is zero based.

Second, whereas the components of a cons cell are expected to be constant, and it is considered to be destructive to change them, the components of an array are expected to be variable, and it is not considered destructive to change them.

LISP arrays can also be printed using functions such as `print` and `princ`, and read by `read`. What is printed is in fact an *s-expression* with a special syntax, not given elsewhere in this chapter, that consists of a #, followed by the number of dimensions in the array (usually 1 or 2), followed by the letter A or a, followed by the array elements in nested lists, where the nesting depth equals the number of dimensions. For example:
1.2. INTERNAL REPRESENTATION OF LISP VALUES

-> (setf v #1a(1 2 3 4)) ; 4 element vector
#(1 2 3 4)
-> (setf m #2a((11 12 13 14) ; 3x4 element matrix
(21 22 23 24) 
(31 32 33 34)))
#2a((11 12 13 14) (21 22 23 24) (31 32 33 34))

If #1a is followed immediately by a left parenthesis, it may be abbreviated to just #, and this is done on output when v is printed above.

The number of dimensions in an array and the size of each dimension are components of the array object that may be read by the array-dimensions function:

-> (array-dimensions v) ; v is as in above example.
(4)
-> (array-dimensions m) ; m is as in above example.
(3 4)

Here array-dimensions returns a list of the dimension sizes.

The number of dimensions in an array, called the array rank is not changeable once the array is created. The sizes of the dimensions are also not changeable unless the array is created in a way that indicates they should be changeable, which we do not do in this course.

The elements of the array are denoted by the aref function:

-> (aref v 3) ; 3 denotes fourth element
4
-> (setf (aref v 1) 2222) ; 1 denotes second element
2222
-> v
#(1 2222 3 4)
-> (aref m 1 3) ; 2nd row, 4th column
24

For aref the indexing is zero based: the index value 0 denotes the first element of a vector, first row or first column of a matrix, etc. Note that aref may be used with setf to set array elements.

LISP arrays can also be created by the make-array function: see Array Structure.

1.2.6 LISP Equality Testing Functions

Now that we know how values are represented in LISP, we can say something about how the different LISP equality functions, eq, eql, equal, and = work.

The eq function just compares the pointers that point at the values being compared. It works for symbols, because no two symbols in memory have the same name. It does not work for very
large integers, because two different large integer values in memory may have the same value as integers. Eq can tell you whether two variables point at the same cons cell, but if not, says nothing about whether the two cons cells have the same cars and cdrs. Thus:

```
-> (eq 'x 'x)
T
-> (eq 12345678901234567890 12345678901234567890)
NIL
-> (eq 1 1.0)
NIL
-> (eq (cons 'a 'b) (cons 'a 'b))
NIL
```

The eql function works like eq unless both values are numbers, in which case it compares the values pointed at, and not the pointers. The complete values are compared, including the type component, so the numbers must have the same type to be eql. Thus:

```
-> (eql 'x 'x)
T
-> (eql 12345678901234567890 12345678901234567890)
T
-> (eql 1 1.0)
NIL
-> (eql (cons 'a 'b) (cons 'a 'b))
NIL
```

The equal function works like eql unless both values are cons cells, in which case equal compares the cars and cdrs of these cells by calling itself recursively. Thus:

```
-> (equal 'x 'x)
T
-> (equal 12345678901234567890 12345678901234567890)
T
-> (equal 1 1.0)
NIL
-> (equal (cons 'a 'b) (cons 'a 'b))
T
```

The = function can only be applied to numbers, and will check for equality by converting number types if necessary. Thus:
1.3. SUMMARY OF LISP

In this section we will summarize the part of COMMONLISP used in this course. This section can be used for review, and may possibly be of help in learning COMMONLISP, but it is not intended as a replacement for a COMMONLISP textbook.

On the other hand, this section does help define the subset of COMMONLISP used in this course, and reviews details of variables, functions, and evaluation that students should know by the end of the course. However, students should realize they have the entire course to learn all these details, and are not responsible near the beginning of the course for details not covered in early lectures.

1.3.1 LISP Variables

In LISP there are two kinds of symbols: special symbols and non-special symbols (special symbols are not to be confused with special forms, which are different). Symbols are declared to be special by defvar statements. Other symbols are non-special.

Whether or not a symbol is special controls how the symbol names variables. Non-special symbols, the most common kind in LISP programs, name variables according to a scheme called lexical binding, which is optimum for symbols naming local variables. Special symbols name variables according to a scheme called dynamic binding, which is optimum for symbols naming global variables.

In order to keep the LISP compiler happy, programmers must use special symbols for global variables and non-special symbols for local variables. The same symbol should not be used to name both global and local variables: therefore COMMONLISP follows the convention that global variables are named by symbols that begin and end with the character * (e.g. *pretty-print*), whereas local variables are named by other symbols.

However, when using just the interpreter (e.g. to experiment with the language or run tests), non-special symbols may be used to name global variables, and the *...* naming convention is often not followed.
We will explain the above statements in more detail in this section, beginning with non-special symbols and lexical binding, continuing with special symbols and dynamic binding, and ending with a discussion of the differences between special and non-special symbols and lexical and dynamic binding.

1.3.1.1 Non-Special Symbols and Lexical Binding

In LISP a non-special symbol (one not declared by `defvar`) can name one global variable and any number of local variables according to a scheme called lexical binding. For example:

```
; Here x is not a special symbol.
(set 'x 1) ; sets global x
(print (eval 'x)) ; prints 1, global x
(let ((x 2)) ; creates local x
   (print x) ; prints 2, innermost local x
   (print (eval 'x)) ; prints 1, global x
   (setf x 22) ; changes innermost local x from 2 to 22
   (set 'x 11) ; changes global x from 1 to 11
   (let ((x 3)) ; create a second local x
      (print x) ; prints 3, innermost local x
      (print (eval 'x)) ; prints 11, global x
      (setf x 33) ; changes innermost local x from 3 to 33
      (set 'x 111) ; changes global x from 11 to 111
   ) ; destroy the second local x
   (print x) ; prints 22, innermost local x
   (print (eval 'x)) ; prints 111, global x
) ; destroy the first local x
```

In this example, `setf x` sets the innermost local (variable named) x; and evaluating just x by itself returns the value of the innermost local x. Similarly `set 'x` sets the global (variable named) x, and `eval 'x` returns the value of the global x.

If there is no innermost local variable named x, `setf x` will set the global variable named x, and evaluating just x by itself will return the value of this global variable. The interpreter never considers these usages to be an error, but the compiler considers them to be in error because the variable x is not special.

Thus using `setf x` to set the value of a non-special global variable, and using x to read that value, is often done when interacting with the LISP interpreter, but should not be done in a proper program, which might be compiled sometime.

A major advantage of lexical binding is that many separate local variables can be introduced in
a large program without having to insist that these local variables all have different names. Thus
different programmers can write different parts of the large program without consulting each other
concerning the names of their local variables.

1.3.1.2 Special Symbols and Dynamic Binding

Special symbols (those declared by `defvar`) name variables according to a scheme called dynamic binding. If `*x*` is a special symbol, `*x*` names only one variable, the global variable, and there can be no local variables named `*x*`. When a special symbol is used as the name of a function argument, or is defined by a `let` or similar statement, a new local variable is not created. Instead, the current value of the global variable named by the symbol is saved, and that global variable is given a new value. When the function or `let` statement ends, the saved value is restored to the global variable. There is no way at all to get the saved value or change it before it is restored.

For example:

```lisp
(defvar *x* 1) ; declares *x* special, sets global *x*
(print *x*) ; prints 1, global *x*
(setf *x* 11) ; changes global *x* from 1 to 11
(let ((*x* 2)) ; saves the current value of global *x*,
    (print *x*) ; prints 2, global *x*
    (print (eval '*x*)) ; prints 2, global *x*
    (setf *x* 111)) ; changes global *x* from 11 to 111
                   ; end of let statement:
                   ; sets global *x* to saved val., 11
(print *x*) ; prints 11, global *x*
```

If `*x*` is a special symbol, `set *x*` and `eval *x*` will work just as if `*x*` were not a special symbol, and are equivalent to `setf *x*` and `*x*` by itself. The compiler never considers use of `setf *x*` or `*x*` by itself to be an error if `*x*` is a special symbol.

Because it is important to know whether a symbol is special or not, COMMONLISP follows the convention that the names of special symbols begin and end with *. For example, the global variable named by the special symbol `*print-pretty*` is used in COMMONLISP to tell the `print` function to print lists in a special indented fashion that is easier to read.

Note that using `*` at the ends of a special symbol name is just a convention. The above example and discussion would work fine if we replaced `*x*` by `x`. The place where difficulty would arise, however, is if two programmers were working together on a large program, and one declared a symbol special in order to name a global variable while the other tried to use the same symbol to name a local variable. This would simply not work, and is the reason for the `*...*` convention.
No problem arises, however, if the two programmers use the same non-special symbol to name different local variables.

A major advantage of dynamic binding is that it is easy to temporarily replace the value of a global variable, by using a `let` statement or something similar. Thus `*print-pretty*` could be temporarily set to `t`, for example, while some printing was being done, and then restored to its previous value, whatever that was. The code to do this would look like:

```
(let ((*pretty-print* t))
  ... do some printing ...
)
```

### 1.3.1.3 Special vs. Non-Special Symbols

From the point of view of a beginning programmer, global variables should always be declared with `defvar` and be named by special symbols that begin and end with `*`, whereas local variables should always be named by non-special symbols that begin or end with some character other than `*`. This convention should keep the compiler and the rest of the world happy.

The beginning programmer should understand that when he uses a non-special symbol as the name of a variable being defined by a `let` statement, or as an argument name in a function definition, he is creating a new local variable; whereas if he uses a special symbol in the same place, he is not creating a new variable, but is instead saving the previous value of the global variable and giving the global variable a temporary new value.

### 1.3.1.4 Lexical vs. Dynamic Binding

The difference between lexical and dynamic binding is essentially the following correspondence:

<table>
<thead>
<tr>
<th>Generic Operation</th>
<th>Lexical Binding Operation</th>
<th>Dynamic Binding Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Variable</td>
<td>Create Local Variable</td>
<td>Save Global Variable Value</td>
</tr>
<tr>
<td>Destroy Variable</td>
<td>Destroy Local Variable</td>
<td>Restore Global Variable Value</td>
</tr>
</tbody>
</table>

For example, when we say the `let` creates a variable, then if the name of the variable is not a special symbol, a local variable is created, but if the name is a special symbol, the global variable’s value is saved.

To bring out the difference between lexical and dynamic binding, consider the interpreter dialog:
1.3. SUMMARY OF LISP

-> ; Here x is not a special symbol
   (setq x 99) ; set global x
   99
-> (defun getx () x) ; getx gets global x always
   GETX
-> (getx) 99

-> (let ((x 88)) ; make new local x equal 88
   (getx)) ; returns the global value 99
   99

-> ; declare *y* special,
   (defvar *y* 99) ; set global *y* to 99
   Y
-> (defun gety () *y*) ; gety gets global *y* always
   GETY
-> (gety) 99

-> (let ((*y* 88)) ; save old value of *y*, 99,
   (gety)) ; and set global *y* to 88
   88 ; returns the global value 88
   ; restore global *y* to
   ; saved value, 99

Note that the definition of getx is legal in interpreted code, but will be considered an error in compiled code because x is used by itself to refer to the global variable named by x, and x is not a special symbol. On the other hand, the compiler will be perfectly happy with gety.

1.3.2 LISP Functions, Macros, and Special Forms

In LISP a symbol names one global function definition and any number of local function definitions. For example:
(defun foo (x) (+ x 1)) ; defines global foo
(print (foo 0)) ; prints 1, calling global foo
(print (funcall #'foo 0)) ; prints 1, calling global foo
(print (funcall (symbol-function 'foo) 0)) ; prints 1, calling global foo
(flet ((foo (x) (+ x 2))) ; creates local foo
  (print (foo 0)) ; prints 2, calling innermost
    ; local foo
  (print (funcall (symbol-function 'foo) 0))) ; prints 1, calling global foo
(print (funcall #'foo 0)) ; prints 2, calling innermost
  ; local foo
)

Using foo as the first element of a list calls the innermost local function definition of foo, if any, or the global function definition if there is no innermost local definition.

Symbol-function is a function that always returns the global function definition of a symbol.

#'foo returns the innermost local function definition of foo, if any, or the global function definition if there is no innermost local definition.

Each function definition of a symbol can be either an ordinary function or a macro (but not both). To make the global definition a macro, defmacro is used in place of defun. To make a local definition a macro, macrolet is used in place of flet. Macros are discussed further below.

Defun can be used to define recursive functions. Flet cannot be, because the function names it defines are not visible to the function definitions it supplies. Labels is an alternative that is just like flet, but permits its function names to be used in its function definitions to define recursive functions. Thus:

(flet ((foo (x) (+ x 1))) ; This foo adds 1.
  (flet ((foo (x) (+ 1 (foo x)))) ; This foo adds 2.
    ; In (foo x) the first foo,
    ; which adds only 1,
    ; is called.
  (labels ((foo (x) (+ 1 (foo x)))) ; This foo is recursive
    ; and loops forever.
    . . . )))

When a function is called, its arguments are all evaluated first, before the function executes.
1.3. SUMMARY OF LISP

Then the function executes, and returns a result value.

When a macro is called, its arguments are not evaluated. The macro is called like a function, but with unevaluated arguments. The macro executes and returns a result called the macro expansion. This macro expansion, which must be an expression that can itself be evaluated, is then evaluated as if it had been originally written in place of the macro call, to produce the final result of that macro call.

For example:

```
-> (defmacro twice (f c x) (list f c (list f c x)))
TWICE
-> (macroexpand '(twice + 3 88))
(+ 3 (+ 3 88))
-> (twice + 3 88)
94
```

Here the function macroexpand is applied to an expression to find out what its macro expansion is.

In COMMONLISP, a few symbols have function definitions that are neither functions nor macros, but are special forms instead. New special forms cannot be defined, nor can the function definitions of special forms be changed. Special forms are built into the language in a fundamental way.

The following are the COMMONLISP special forms used in this course:

```
catch  flet  function (#')  if
labels  let  let*  macrolet
progn  quote (')  setq  throw
unwind-protect
```

Special forms are like macros, in that they do not evaluate at least some of their arguments, or they delay evaluation of arguments until some appropriate time. Many other COMMONLISP operations are defined as macros that expand into special forms. Some examples are cond, defun, and do. You can find out what these macros expand into by using macroexpand, as in the following examples:

```
-> (macroexpand '(defun x (y) z))
(PROGN (SETF (SYMBOL-FUNCTION 'X) #'(LAMBDA (Y) (BLOCK X Z))) 'X)
-> (macroexpand '(cond ((= x y) z) ((< x y) w) (t v)))
(IF (= X Y) Z (IF (< X Y) W V))
```

Note that the results are not necessarily simple. For example, block is a special form not used in this course. (block x z) is equivalent to (progn z) except that inside the block expression a (return-from x e) can be used to stop executing code in the block expression and
return the value $e$ from the block expression; \texttt{return-from} is another special form not used in this course.

In function definitions such as
\[
\texttt{-> (defun twice \{f \} \{f (f \ f)\})}
\]
\texttt{TWICE}
\[
\texttt{-> (setq f #\(\texttt{lambda} \{y\} (+ 1 y))\)}
\]
\[
\texttt{(LAMBDA-CLOSURE () () () \{Y\} (+ 1 Y))}
\]
the lists of variable names, here \(f \ x\) and \(y\), are called \textit{lambda lists}. COMMONLISP permits a number of options in the lambda list, but the only one we will use in this course is the \texttt{&rest} option. This option permits functions and macros to be written that take indeterminate numbers of arguments:

\[
\texttt{-> ; Function to return the sum of the squares of its arguments:}
\texttt{(defun sum-squares \{x \&rest ylist\})}
\texttt{(let \{\(\texttt{sum} \times x\)\})}
\texttt{(dolist \{y\} ylist \{sum\})}
\texttt{(setq sum (+ sum \times y))})
\]
\texttt{SUM-SQUARES}
\[
\texttt{-> (sum-squares 3 4) ; binds 3 to \texttt{x} and \(4\) to \texttt{ylist}}
\texttt{25}
\]
\[
\texttt{-> (sum-squares 3 4 10) ; binds 3 to \texttt{x} and \(4\ 10\) to \texttt{ylist}}
\texttt{125}
\]
\[
\texttt{-> (sum-squares 3) ; binds 3 to \texttt{x} and \emptyset \texttt{ylist}}
\texttt{9}
\]

When a function or macro is called, the actual arguments are bound to the argument names. If the last argument name in the lambda list is preceded by \texttt{&rest}, then instead of binding to it the last argument, LISP binds to it a list of all the remaining arguments. There may be zero or more remaining arguments. Thus the call \texttt{(sum-squares 3 4 10)} binds \(4\ 10\) to \texttt{ylist}.

\subsection*{1.3.3 How LISP Evaluates S-Expressions}

(a) If the s-expression is a number, LISP returns the number.

(b) If the s-expression is a symbol, LISP returns the value of the innermost local variable named by the symbol. If there is no innermost local variable named by the symbol, LISP returns the value of the global variable named by the symbol (but the compiler considers this an error if the symbol is not a special symbol).

(c) If the s-expression is a list, LISP requires that the first element of the list be a function
specifier. The remainder of the list consists of arguments to which the function is to be applied.

If the function specifier is a non-special-form symbol, the innermost local function definition named by that symbol is used, or if none, then the global function definition named by that symbol is used.

(c1) If the function specifier is a non-special-form symbol and the function definition to be used defines a function (either builtin, like car, or user defined, as by defun), the arguments are evaluated first, before the function is called. Then the function is called, and the value returned by the function is the value of the s-expression.

(c2) If the function specifier is a non-special-form symbol and the function definition to be used defines a macro, (either builtin, like cond, or user defined by defmacro), the macro is called like a function but with the arguments unevaluated. The value returned by the macro, the macro expansion, replaces the original s-expression, and is evaluated in place of that expression.

You can find the macro expansion of an s-expression via the macroexpand function:

```
-> (macroexpand '(cond (x 1) (t 2)))
(IF X 1 2)
```

(c3) If the function specifier is a symbol naming a special form (e.g. quote or if), the s-expression is specially evaluated. Some arguments may be evaluated first, some may be evaluated later in a specific order, and other arguments may not be evaluated at all. COMMONLISP does not allow users to define special forms, and therefore has only a limited number of possible special forms. (Note that the many COMMONLISP macros which expand into special forms behave like special forms in choosing when to evaluate their arguments.)

Example:

```
-> '(* 2 33)
(* 2 33)
-> (setq x 1)
1
-> (if (= x 2) (setf x 33))
NIL
-> x
1
```
(c4) If the function specifier is a list beginning with the symbol `lambda`, this list is treated as a function definition, the arguments are evaluated first, then the function is called, and the value it returns is the value of the s-expression.

Example:

```
-> ((lambda (x) (* 2 x)) 11)
22
```

### 1.3.4 Builtin Functions, Macros, and Special Forms

The material in this section is to be used as a reference, and includes information that is not explained properly until later in this book.

Unless stated otherwise, the functions described below are normal functions, and not special forms, macros, destructive functions, or global variables.

* means “zero-or-more”, as in our grammars. The multiplication operator * is also used, occasionally, in a slightly different typeface.

Words in italics represent arguments. If they are underlined, then they are not evaluated: otherwise they are evaluated, although for special forms and macros this evaluation may be conditional (e.g. if). Arguments that must be of a particular type are given names that obviously indicate this, such as symbol, number, or list.

The term “create variable” means to create a local variable if the name of the variable is not a special symbol, or to save the global variable value if the name is a special symbol. The term “destroy variable” means to destroy a local variable if the name of the variable is not a special symbol, or to restore the value of the global variable if the name is a special symbol.

It is implicit that any special form or macro that creates a variable or local function definition will destroy the variable or local function definition when the special form or macro terminates.

#### 1.3.4.1 List Structure

(car list) Returns the first element of list.

(cdr list) Returns list with the first element removed.

(cons car-value cdr-value) Returns the list with the specified car and cdr.

```lisp
-> (cons (car '(a (b))) (cdr '(d e)))
(A E)
```
1.3. SUMMARY OF LISP

1.3.4.2 Symbol Structure

(eval symbol) Returns the value of the global variable named by symbol.

(symbol-value symbol) Ditto.

(set symbol value) Sets the global variable named by symbol to have the given value and returns this value.

(setf (symbol-value symbol) value) Macro. Ditto.

(boundp symbol) True iff the global variable named by symbol has a value.

(makunbound symbol) Make the global variable named by symbol so that it has no value.

(defun fun (var* &rest rest-var) expression*) Macro. Defines the global function named by the symbol fun. See Defining and Using Functions below for details.

(defmacro fun (var* &rest rest-var) expression*) Macro. Defines the global function named by the symbol fun, but defines it to be a macro and not a function (the global function definition of a symbol must be either a macro or a function, but not both). See Defining and Using Functions below for details.

(symbol-function symbol) Returns the value of the global function definition named by symbol. This value is implementation dependent if it is a macro definition or if the symbol names a special form.

(fboundp symbol) True iff the global function definition named by symbol has a value (either a function or macro definition), or if symbol is a special form.

(setf (symbol-function symbol) value) Macro. Sets the global function definition named by symbol to have the given value, and returns this value.

(symbol-plist symbol) Returns the value of the property list named by symbol.

(setf (symbol-plist symbol) value) Macro. Sets the property list named by symbol to have the given value, and returns this value.
1.3.4.3 Array Structure

(make-array dimension-size-list :initial-element value)
Creates and returns an array whose dimension sizes are given by the (non-negative integer) elements of the dimension-size-list. Each element is initialized to value. Omitting :initial-element and value causes each element to be initialized to nil.

(array-dimensions array) Returns a list whose elements are the sizes of the dimensions of array.

(aref array index1 index2 ...) Returns the element of array selected by the indexes. There must be as many indexes as there are dimensions of the array. The index value 0 selects the first row, column, etc. of the array: i.e., the indexing is “base 0”.

(setf (aref array index1 index2 ...) value) Stores value in the element of array selected by the indexes, and returns value. See above for description of the indexes.

-> (boundp 'x)
NIL
-> (set 'x 55)
55
-> (boundp 'x)
T
-> (eval 'x)
55
-> (defun foo (x) (* 2 x))
FOO
-> (symbol-function 'foo)
(LAMBDA (X) (* 2 X))
1.3. SUMMARY OF LISP

1.3.4.4 Local Variables

In this section we describe just the let and let* special forms that create variables. However, there are many other special forms and macros that create variables. For example, the do, dolist, and dotimes macros. Also, when a function or macro is called, a variable is created for each argument. These variables are assigned names by the defun, lambda, flet, or labels which created the function; or by the demacro or macrolet which created the macro.

Note that only non-special symbols (those not mentioned in a defvar statement) can name local variables. Special forms such as let, when used on special symbols, actually save and restore the values of the global variables named by the symbols, instead of creating local variables. See Builtin Functions, Macros, and Special Forms.

(let ((var value)*)) Special form. Each symbol var names a variable to be created. Creates the vars and assigns to them the given values. Then evaluates the expressions from left to right. The values are all evaluated before any variables are created, so no value can refer to any var. The value of the last expression is the value of let.

(let* ((var value)*)) Special form. Ditto but the variables are created one at a time so each value can refer to any previous var.

var Here var is a symbol naming a variable by itself. The value of the innermost local variable named by var is returned.

(setf var value) Macro. Assigns a new value to the innermost local variable named by the symbol var and returns this value.

(let ((x 1) (y 2))
  ; Now innermost x = 1 and innermost y = 2
  (let ((x (+ x 10)) (y (+ x 20)))
    ; Now innermost x = 11 and innermost y = 21
    (let* ((x (+ x 100)) (y (+ x 200)))
      ; Now innermost x = 111 and innermost y = 311
      (setf x 55)
      ; Now innermost x = 55 and innermost y = 311
    )
    ; Now innermost x = 11 and innermost y = 21
  )
  ; Now innermost x = 1 and innermost y = 2
)
1.3.4.5 Global Variables

A symbol used to name a global variable should be declared to be a special symbol by `defvar`, and cannot be used to name local variables. To enforce this rule more easily, it is a convention in COMMONLISP to give global variables names that begin and end with * (e.g. *print-pretty*).

If you use a special symbol to name a variable defined by `let`, `defun`, etc., then instead of creating a local variable, the value of the global variable named by the symbol will be saved for restoration at the end of the `let` or function, and the global variable will be given a new value and used throughout the body of the `let` or function. Later, instead of destroying the local variable, the global variable will be reset to the saved value.

**(defvar var value)** Macro. Declares the symbol `var` to be a special symbol and sets the global variable it names to the `value`.

`var` Here `var` is a special symbol naming a global variable by itself. The value of the variable is returned.

**(setq var value)** Macro. Assigns a new `value` to the global variable named by the special symbol `var`, and returns this value.

```
-> (defvar *x* 55) ; Declare *x* special, set it to 55.
*X*
-> *x*
55
-> (setq *x* 9)
9
-> (let ((*x* 22)) ; Save 9, set global *x* to 22.
  (eval ' *x* ) ; Return global *x* equals 22.
  ; (Were *x* not special,
  ;  global *x* would still be 9.)
  ) ; End let: restore 9 to *x*.
22
```

1.3.4.6 Defining and Using Functions

**(function expression)** This is usually abbreviated by `#' expression`. See `#` below.

`#'(lambda (var*) expression*)` Special form. Evaluates to a function that may be stored in a variable and executed by `funcall` or `apply` (see below). Each `var` is a symbol naming a
variable created at the beginning of the function that is bound to one of the function arguments during execution of the function. The expressions are then evaluated in order to evaluate the function, with the value of the last being returned as the value of the function.

```lisp
#'(lambda (** &rest ***) expression*)
```

Ditto, but when the function is called, all the argument values left over after binding values to the ** are made into a list which is bound to ***.

```lisp
(defun fun (**) expression*)
```

```lisp
(defun fun (** &rest ***) expression*)
```

Macro. Defines the global function named by the symbol fun. The function is as for #’ (lambda (** expression*)) or #’ (lambda (** &rest ***) expression*) above.

```lisp
(defmacro fun (**) expression*)
```

```lisp
(defmacro fun (** &rest ***) expression*)
```

Macro. Ditto but defines a macro and not a function. The macro is called with unevaluated arguments, and the expression it returns is then evaluated as if it had been originally written in place of the macro call.

```lisp
(flet ((fun (**) expression*)*)
```

```lisp
(flet ((fun (** &rest ***) expression*)*)
```

Special form. Each fun is a symbol naming a new function to be defined. Creates new local function definitions named by the fun symbols, evaluates the statements in order, and returns the value of the last statement evaluated.

Each function definition is computed as for (lambda (** expression*)) or (lambda (** &rest ***) expression*) above. The names of the functions, the fun symbols, may not be used in the definitions of the functions to refer to these functions. The function definitions are computed before any of the funs are defined.

```lisp
(labels ((fun (**) expression*)*)
```

```lisp
(labels ((fun (** &rest ***) expression*)*)
```

Special form. Just like flet except that the names fun of the functions may be used in the definitions of the functions to refer to the functions, thus allowing mutually recursive functions to be defined. I.e., labels is “recursive-flet”.

1.3. SUMMARY OF LISP

41
(macrolet ((fun (var*) expression*)*)
  statement*)
(macrolet ((fun (var* &rest rest-var) expression*)*)
  statement*)

Special form. Just like let except that macros are defined instead of functions. Locally defined variables, functions, and macros are not visible to the macro definition code (the expressions), but are visible to the macro expansion.

#'symbol Special form. Returns the innermost local function definition of symbol, if any, or the global function definition of symbol if there is no local definition. Value is undefined if definition is that of a macro or special form.

(symbol-function symbol) Returns the global function definition of symbol. Value is undefined if definition is that of a macro or special form.

(funcall function arg*) Returns the result of applying function to the arguments arg*. The number of arguments must match the number required/allowed by function.

(apply function list) Returns the result of applying function to arguments that are the elements of list. The length of list should equal the number of arguments required/allowed by function.

(mapcar function list*) Returns a list whose k’th element is the result of applying function to arguments which are the k’th elements of the lists. The number of lists must equal the number of arguments required/allowed by function. The result is of the same length as the shortest list.

(macroexpand expression) Performs macro expansion recursively on expression, until the result is not a macro call, and returns the result.

-> (apply #'+ '(1 4 5))
10
-> (funcall #'+ 1 4 5)
10
-> (mapcar #'(lambda (x y z) (+ x y z))
  '(1 2 3) '(4 1 3) '(5 3))
(10 6)
-> (macroexpand '(cond (x 8) (t 9)))
(IF x 8 9)
1.3. SUMMARY OF LISP

1.3.4.7 Predicates

Nil signifies false; any non-nil value signifies true. Functions that return true or false generally return t for true.

The descriptions below state when the predicate functions return true. It is implicit in these descriptions that when the predicate functions do not return true, they return nil.

(atom value) Returns t if value is an atom (nil is an atom).

(consp value) Returns t if value is a cons cell (nil is not a cons cell).

(listp value) Returns t if value is a list (nil is a list).

(null value) Returns t if value is nil.

(not value) Returns t if value is false (i.e. nil).

(> number1 number2 number*) Returns t if number1 > number2 > ....

Similarly <, <=, >=, =, /= (not equal).

The not-equal operator /= requires that none of the numbers equal each other. Note that these operators, = and /= in particular, are only applicable to numbers.

(zerop number) Returns t if number = 0.

(evenp integer) Returns t if integer is even.

(oddp integer) Returns t if integer is odd.

(equal value1 value2) Returns t if value1 and value2 look the same when printed.

(eql value1 value2) Returns t if value1 and value2 are the same symbol, the same cons cell (not different cons cells with the same car and cdr), or numbers with both the same type and same value. Faster than equal.

(eq value1 value2) Returns t if value1 and value2 are the same symbol or the same cons cell (not different cons cells with the same car and cdr). Should not be used on numbers (may or may not return reasonable results, depending on implementation). Much faster than eql in compiled code.

(member object list) Checks if object is eql to an element of list. If true, returns all of list after and including object.
(member object list :test function) Ditto but uses function instead of eql. The most common
function is `equal`.

-> (eq (cons 'a nil) (cons 'a nil))
NIL
-> (equal (cons 'a nil) (cons 'a nil))
T
-> (eql 0 0.0)
NIL
-> (= 0 0.0)
T
-> (member 'a '(b c a (d)))
(A (D))
-> (member '(b) '(a (b) d))
NIL
-> (member '(b) '(a (b) c) :test #'equal)
((B) C)

1.3.4.8 Arithmetic

In general, if only exact numbers — integers or ratios — are involved in calculating a function
value, the result is an exact number. If any inexact number — a floating point number — is in-
volved, all numbers are converted to the largest floating point number type involved before doing
the calculation. However, transcendental functions like `sqrt` and `expt` with a non-integer expo-
nenent may produce floating point results even when applied to perfect squares (or they may produce
exact results when possible, if the implementation chooses).

(+ number*) Adds all the numbers together.

(* number*) Multiplies all the numbers together.

(- number) Returns the negative of number.

(- number number number*) Subtracts all numbers but the first from the first number.

(1+ number) Returns number plus 1.

(1- number) Returns number minus 1.
1.3. SUMMARY OF LISP

(truncate number number) Divides the first number by the second number, and rounds the result toward zero to an integer. If the numbers are integers this gives the same result as division of integers as taught in U.S. schools.

For positive integers, this is the same as / division in the C language. For integers of different signs, this is the same as / division in C on most computers, but the C language reserves the right to define / differently (to round like floor) on some computers.

(rem number number) Returns the remainder from the truncate operation above. If the numbers are integers this gives the same result as the remainder from division of integers as taught in U.S. schools.

For positive integers, this is the same as the % operator in the C language. For integers of different signs, this is the same as the % operator in C on most computers, but the C language reserves the right to define % differently (to be like mod below) on some computers.

(/ number number number*) Divides the first number by all the other numbers. If all numbers are exact, the result is in general a ratio and not an integer.

(floor number) Rounds number downward to an integer.

(ceiling number) Rounds number upward to an integer.

(round number) Rounds number to the nearest integer. If there are two nearest integers, the even one is chosen.

(mod number1 number2) Returns number1 modulo number2, that is, the remainder upon dividing number1 by number2 and rounding down (like floor above) to get an integer.

(random integer) Returns a random integer between 0 and one less than the given strictly-positive integer argument, inclusive.

(sqrt number) Returns the square root of number.

(expt number1 number2) Returns the number1 taken to the number2 power.

1.3.4.9 Lists and Evaluation

(quote expression) Special form. This is usually abbreviated by ‘expression. See ‘ below.

’expression Special form. Returns expression without evaluating it.
CHAPTER 1. THE STRUCTURE OF LISP

(eval expression) Evaluates the given expression, and returns the result. In effect, expression is evaluated twice, once before eval is applied to it (because it’s an argument) and once by eval.

(list value*) Returns the list (value*).

(list* first value* rest) Returns the dotted list (first value*. rest).

(caar list) and also cadr, caddr, cddr, . . . , cddddr.

(cdr list) is equivalent to (car (car list)). (cadr list) is equivalent to (car (cdr list)). And so forth, with up to 4 a’s or d’s permitted in the function name.

(nth n list) Returns the nth element of list, starting count from 0. The argument n must be a non-negative integer; if it is equal to or greater than the length of the list, nil is returned.

(length list) Returns the length of list.

(reverse list) Returns the list with its elements reversed.

(append list*) Returns the list made by concatenating all the lists together.

(remove value list :test function) Returns list with all elements equal to value removed, where function is used to test for equality. Often function is #'equal. If :test and function are omitted, #'eql is used.

(assoc value a-list :test function) A-list must be a list of cons cells (lists or dotted-pairs). Assoc returns the first cons cell in a-list whose car is equal to value, where function is used to check for equality. Often function is #'equal. If :test and function are omitted, #'eql is used.

(sort (copy-list list) function) Returns list sorted so that each two successive elements satisfy the test specified by function. E.g. if function is #'< and the list is a list of numbers, the numbers will be sorted in ascending order. Sort by itself is a destructive function, and should be used with copy-list as indicated to make a non-destructive sort.
1.3. SUMMARY OF LISP

-> (list '+ '1 '2 '3)
(+ 1 2 3)

-> (append (list '+ '1 '2 '3 ) '(4))
(+ 1 2 3 4)

-> (quote (append (list '+ '1 '2 '3 ) '(4)))
'(APPEND (LIST '+ '1 '2 '3 ) '(4))

-> (eval (quote (append (list '+ '1 '2 '3 ) '(4))))
(+ 1 2 3 4)

-> (eval (eval (quote (append (list '+ '1 '2 '3 ) '(4)))))
10

-> (setf x (remove 1 '(3 2 1 2 3 2 1 2 3)))
(3 2 2 3 2 2 3)

-> (list* 1 1 x)
(1 1 3 2 2 3 2 2 3)

1.3.4.10 Property Lists

(get symbol key-value) The property list of symbol is searched for a key/value pair whose key is eq to key-value, and the pair’s value is returned. If no pair is found, nil is returned.

(setf (get symbol key-value) value)
The property list of symbol is searched for a key/value pair whose key is eq to key-value, and the pair’s value is set to value. If no pair is found, one is added. In any case, value is returned.

-> (setf (get 'cs51 'professor) '(harry lewis))
(HARRY LEWIS)

-> (get 'cs51 'professor)
(HARRY LEWIS)

-> (get 'cs51 'tfs)
NIL

1.3.4.11 Conditionals and Looping

(cond (test expression*)*) Macro. Evaluates the tests in order until it finds one that returns true. Then evaluates the list of expressions immediately following the true test, returning the value of the last. If there are no expressions following the test, the value of the true test is returned.
CHAPTER 1. THE STRUCTURE OF LISP

Does not evaluate any test after the first that returns true, or any expression that immediately follows any test other than the first test that returns true. If no tests return true, cond returns nil.

```lisp
-> (cond ((= 1 2) '(one equals two))
     (t '(one is not equal to two)))
(ONE IS NOT EQUAL TO TWO)
```

(if test then-expression else-expression) Special form. Evaluates test first. If true, evaluates then-expression and returns its value, but does not evaluate else-expression. If test evaluates to false, evaluates else-expression and returns its value, but does not evaluate then-expression. The else-expression can be omitted, in which case it is as if else-expression were nil.

```lisp
-> (if t (+ 1 2) (= 1 1) (* 2 3))
6
```

(and expression*) Macro. The expressions are evaluated from left to right, and evaluation stops when either (1) some expression evaluates to false (nil), in which case nil is returned, or (2) none of the expressions evaluate to false, in which case the value of the last expression is returned (and this is non-nil and therefore represents true). Expressions after the first expression that evaluates to false are not evaluated.

```lisp
-> (and t (+ 1 2) (= 1 1) (* 2 3))
6
```

```lisp
-> (and t
     (+ 1 2)
     (= 1 2)
     (this stuff doesn’t matter —
      evaluation stops with (= 1 2)))
NIL
```

(or expression*) Macro. Similar to and. The expressions are evaluated from left to right, and evaluation stops when either (1) some expression evaluates to false (nil), in which case the value of this expression is returned, or (2) none of the expressions evaluates to true, in which case nil is returned. Expressions after the first expression that evaluates to true are not evaluated.

```lisp
-> (or (+ 1 2) (= 1 1) (* 2 3))
6
```

(do ((var initial update)*)
    (end-condition result*)
    expression*)
Macro. Performs iteration. Initially (1) evaluates the initial value expressions, (2) creates a new variable for each symbol var and assigns the associated initial value, (3) evaluates the
end-condition expression, and if end-condition evaluates to true, evaluates all result expressions in order and returns the value of the last as the result of the do (if there are no result expressions the value of end-condition is returned).

If end-condition was false, then (4) the expressions are evaluated in order, (5) the updates are evaluated in order, (6) the values of the updates are assigned to the vars, and execution continues with (3) above.

The var variables are local to the do statement, and are visible to all of the do statement except the initial expressions. Any update may be omitted, in which case steps (5) and (6) are skipped for that update and its var.

```lisp
-> (defun repeat-10 (arg) ; prints its argument
  ; 10 times
  (do ((count 0 (1+ count))) ; count starts at 0, and
    ; gets incremented each loop
    ((= count 10) t) ; if count = 10 then
      ; end and return t
    (princ arg))) ; in each loop print
  ; the argument

REPEAT-10
-> (repeat-10 'hi)
HIHIHIHIHIHIHIHIHI
T
```

(\texttt{do*} (\texttt{var initial update}*))
\texttt{(end-condition result*)}
\texttt{expression*)}

Macro. Like do, but each var is created immediately after evaluating its initial value, so it is visible to the initial expressions of following vars, and each var is assigned its initial and update values as soon as these have been evaluated, so these values are available to the initial or update expressions of following vars.

(\texttt{dolist} (\texttt{var list return-value) expression*}) Macro. First list is evaluated. Then the variable named by the symbol var is created and iteratively assigned each element of the list. For each element, the expressions are evaluated in order. Finally, return-value is evaluated (with var set to nil) and returned.
-> (setf a 3)
3
-> (setf b 6)
6
-> (dolist (x '(a b) '(done)) (princ (eval x)))
36
(DONE)

(dotimes (var count return-value) expression*) Macro. First count is evaluated: it must be an integer. Then the variable named by the symbol var is created and iteratively assigned the increasing integers from 0 through count - 1 (if count is not above zero, there are none). For each integer, the expressions are evaluated in order. Finally return-value is evaluated (with var set to nil) and returned.

1.3.4.12 Catch and Throw

(catch tag-value expression*) Special form. Catch provides a means of quickly getting out of nested recursion or iterative loops. First, tag-value is evaluated to provide a reference name. Then the expressions are evaluated in sequence. If this evaluation goes normally, the value of the last expression is returned. However, if during the evaluation of the expressions a (throw tag-value value) is encountered, execution of the catch form terminates returning value. The tag-value provided by the throw must eq the tag-value of the catch.

(throw tag-value value) Special form. Searches for the most recently started currently executing catch form whose tag-value is eq to that of the throw, and terminates that catch form causing it to return value. It is an error if there is no currently executing catch with a suitable tag-value.

(unwind-protect expression cleanup-expression*) Special form. Guarantees that the cleanup-expressions will be evaluated even if a throw occurs while evaluating expression to a catch that includes the unwind-protect.

Normally, evaluates expression and then the cleanup-expressions in order, and returns the value of expression.

If a throw occurs while evaluating expression, to a catch that includes the unwind-protect, then the cleanup-expressions are evaluated in order, and if they terminate normally, the throw is continued. However, the cleanup-expressions may execute their own throw, which will terminate the original throw and replace it.
1.3. SUMMARY OF LISP

-> (catch 'done
   (unwind-protect
     (dolist (x '((a 1) (b 2) (c 3) (d 4)) nil)
       (if (eq (car x) 'c) (throw 'done (cadr x)))
     (print 'HELLO)))
HELLO
3

1.3.4.13 Input/Output

(terpri) Prints a carriage return and returns nil.

(prinl value) Prints value (which may be a number, symbol, list, or array) in a format that can be read by read to recreate value, and returns value. Note that |Hello There| prints as |Hello There|.

(print value) Same as prinl, except that first a carriage return is printed.

(princ value) Same as prinl, except that escape characters (| and \) are not printed in symbols. E.g. |Hello There| prints as Hello There.

*print-pretty* Global variable. If set to true, print, princ, and prinl print long lists on multiple lines with indentation that shows the list structure. Printing starts on the current line, and each line is subsequently indented to at least the starting column. If set false, the print functions never end the current line no matter how long it may get.

*print-circle* Global Variable. If true, print, princ, and similar functions will print circular lists without looping forever. If false, they will loop indefinitely. It takes considerable computer time to check for circularity, so this global variable is normally false.

(read)
(read nil nil eof-value) Reads the next s-expression from the input and returns it. The form with no arguments signals an error if an end-of-file is encountered. The form with three arguments, the first two being nil, returns the eof-value if an end-of-file is encountered (we will not use non-nil values for the first two arguments in this course).
1.3.4.14 Loading and Compiling

(load symbol)
(load symbol :print t)
   Reads s-expressions from the file named by the symbol and evaluates each s-expression read. Returns T. If :print t is given, the results of each evaluation are printed.

(compile symbol) Compiles the global function definition of symbol.

(compile nil '(lambda (var* &rest rest-var) expression*))
   Compiles the lambda abstraction and returns the compiled function. The second argument can actually be anything that evaluates to a lambda abstraction.

-> (load '|myfile.lsp|) ; Use |'s to protect lower case.
T
-> (defun add1 (l) (mapcar #'1+ l))
ADD1
-> (compile 'add1)
ADD1
-> (add1 '(3 5 7))
(4 6 8)
-> (funcall (compile nil '(lambda (l) (mapcar #'1- l))) '(3 5 7))
(2 4 6)

Both the load and compile functions may print additional information about what they are doing, such as messages indicating the beginning and ending of loading and the end of various compiler passes.

1.3.4.15 Debugging

(assert expression) Expression is evaluated, and if it is not true, an error message is printed indicating that expression is false and the program breaks (see break).

(error "message") An error message is printed containing the unevaluated text string message, and the program breaks (see break). Continuation of execution is not permitted from this program break.

The syntax "message" is that of a COMMONLISP character string type datum. For every other function that we use in this course, a symbol as in ’|message| can be used instead; but not for error.
1.3. SUMMARY OF LISP

(break) The program stops execution and permits the user at his console to type in s-expressions which are evaluated with their results being printed. There is also some implementation dependent way to continue the program and to examine the execution stack of the program.

(pprint value) Just like print but if value is a long list, it is printed on multiple lines with indentation to make the list easy to read.

(trace symbol) Macro. The global function definition of symbol is set to “traced”. A traced function will print out its arguments whenever it is called, and print out its return value whenever it returns. It also prints its recursion depth.

(untrace symbol) Macro. The global function definition of symbol is set to “untraced”. I.e., untrace cancels trace.

(dribble filename-symbol) The LISP interpreter starts dribbling to the named file. Dribbling means that all typed input and output is copied into the file. Dribbling may be stopped by calling dribble with no argument.

(time expression) Macro. The expression is evaluated and the time that it takes to execute is printed. The value of expression is returned.

-> (assert (= 5 5))
NIL
-> (assert (= 5 6))
Correctable error: The assertion (= 5 6) is failed.
Signaled by DO.
If continued:

Broken at CERROR. Type :H for Help.
>>:q ; For IBCL and KCL :q returns to top level.
Top level.
-> (error "A practice error.")
Error: A practice error.
Error signaled by EVAL.

Broken at ERROR. Type :H for Help.
>>:q
Top level.
-> (defvar *x* 9)
*X*

-> (defun addtox (y) (setf *x* (+ *x* y)) (break))
ADDTOX

-> (addtox 1)
Break.

Broken at ADDTOX. Type :H for Help.
>>(pprint *x*)
10
>> ; For IBCL and KCL :r resumes execution
 :r ; right after the break.
NIL

-> (defun addtox2 (y) (setf *x* (+ *x* y)))
ADDTOX2
-> (defun addtox1 (y) (addtox2 y))
ADDTOX1

-> (trace addtox1 addtox2)
(ADDTOX1 ADDTOX2)

-> ; In 1>, 2>, <2, <1,
 ; 1 and 2 are recursion depths,
 ; > gives called function and arguments,
 ; < gives returning function and values
 (addtox1 1)
> 1> (ADDTOX1 1)
 2> (ADDTOX2 1)
 <2 (ADDTOX2 11)
 <1 (ADDTOX1 11)

11

-> (untrace addtox1 addtox2)
(ADDTOX1 ADDTOX2)

-> (time (dotimes (i 1000) (addtox2 1)))
real time : 0.667 secs
run time : 0.233 secs
NIL
1.3. SUMMARY OF LISP

1.3.4.16 List Destructive Functions

(delete value list :test function) Destructive. Deletes all elements equal to value from list, returning the resulting list. Uses function, which is typically #’equal, to determine equality. If :test and function are omitted, uses eql to determine equality.

(nconc list*) Destructive. Concatenates all the lists and returns the resulting list.

(nreverse list) Destructive. Reverses all the elements in list and returns the resulting list.

This function is very useful in the situation where it is convenient to create a list in reverse order. After creation, nreverse can reverse the list to the proper order without the overhead of creating new cons-cells. Since the list has just been created, it is generally only pointed at by one known variable.

(rplaca cons-cell value) Destructive. Stores value in the car of the cons-cell, and returns the cons-cell.

(rplacd cons-cell value) Destructive. Stores value in the cdr of the cons-cell, and returns the cons-cell.

((setf (car cons-cell) value) Destructive. Stores value in the car of the cons-cell, and returns the value.

((setf (cdr cons-cell) value) Destructive. Stores value in the cdr of the cons-cell, and returns the value.

(sort list function) Destructive. Sorts list and returns the result. The list is sorted so that when function is applied to two successive elements it returns true. A typical value for function is #’<.
1.4 LISP Projects

The written problems in these sections should be answered in an ASCII text file named answers.txt that begins with the file header:

{one-line description of file}

File: answers.txt
Assignment: {1, 2, etc.}
Author: {Your name}({your email address})
Version: 1

Here you fill in the bracketed parts. If you happen to submit a second version of the file, increment the version number.

Note that collaboration is not permitted on problems in these sections.

1.4.1 Project: Evaluation of LISP Expressions

1. For each of the following s-expressions, describe the results of evaluation. Assume that each s-expression is typed independently into a LISP interpreter that has just started to run. If the expression is in error, explain the error and how to fix it, but you need not give the actual
1.4. LISP PROJECTS

error message. If there is no error, give both the result of evaluating the expression, and any
side-effects the evaluation has.

Write your answers in an ASCII text file named answers.txt.

(a) (list 'I 'WAS 'SOLVING nil 'AND t)
(b) (append '(hello there) (cons '(I am going) '(to help you)))
(c) (eval (list '+ '(2 3 4)))
(d) (/ 24 5)
(e) (1+ (* '9) + (- 8 7 1))
(f) (setf x (car (caddr (cdr '(a (b (c d) e ((f))))))))
(g) (member 1 (3 5 4 1 6 8))
(h) (member (cons 1 2) (list (cons 3 4) (cons 1 2)))
(i) (and (= 1 1.0) (= 0.5 1/2) (/= 1 2 1) (howareyou))
(j) (let ((x 9)) (cons x nil))
(k) (funcall (setf x #'(lambda () x)))
(l) (mapcar #'mapcar (list #'1+ #'1-) ’((1 2) (3 4) (5 6)))
(m) (nreverse (setf x (list 1 2 3 4 5)))
(n) (funcall #’(lambda (x y) (cons (car x) (cdr y)))
(a b) ’(c d))
(o) (progn (setf x (cons ’a nil)) (rplaca x x))

2. In one or two English sentences, what is different about the following pairs of expressions
when they are evaluated, Write your answers in an ASCII text file named answers.txt.

(a) i. (append (setf y nil) ’(setf z ’(a b c)))
    ii. (append ’(setf y nil) (setf z ’(a b c)))
(b) i. (setf z ’car)
    ii. (setf z #’car)
(c) i. (setf (car x) 9)
    ii. (rplaca x 9)
(d) i. (nreverse (setf x (list 1 2 3 4)))
    ii. (nreverse (setf x ’(1 2 3 4)))
1.4.2 Project: Representation of LISP Expressions

1. For each of the following dotted-pair box diagrams, write out the s-expression that it represents (in simplified form) in an ASCII text file named answers.txt. Recall that simplified form removes excessive dots and parentheses by the means of the following rules:

\[
\begin{align*}
() & \iff \text{nil} \\
(a . \text{nil}) & \iff (a) \\
(a . (b \ c)) & \iff (a \ b \ c)
\end{align*}
\]

(a)  
\[
\text{\includegraphics{diagram_a.png}}
\]

(b)  
\[
\text{\includegraphics{diagram_b.png}}
\]

(c)  
\[
\text{\includegraphics{diagram_c.png}}
\]

(d)  
\[
\text{\includegraphics{diagram_d.png}}
\]
2. For each of the following s-expressions, write out the equivalent dotted-pair box notation in the ASCII text file `answers.txt`. You may use notation such as:

\[
\begin{align*}
\text{car.cdr} & \longrightarrow \text{car.cdr} \longrightarrow \text{car.cdr} \ldots \\
& \quad \mid \\
& \quad \text{Hello} \quad \text{There} \quad \text{car.cdr} \\
& \quad \quad \mid \\
& \quad \quad \mid \\
& \quad \quad \text{My} \quad \text{Dear} \\
& \quad \quad \quad \mid \\
& \quad \quad \quad \text{car.nil} \\
& \quad \quad \quad \mid \\
& \quad \quad \quad \text{Friend}
\end{align*}
\]

(a) (a (b) c)
(b) (a b (c nil (d e) f) g h (i))
(c) ((nil))
(d) (a b (c . d) . (e . f))
Chapter 2

A Tour of LISP

by Jason Abrevaya, Bob Walton, and Harry Lewis

In this chapter we examine ways in which LISP can be used.

2.1 Recursion in LISP

In mathematics, recursion is the primary means of giving a finite definition for the value of a function when the arguments to that function can be arbitrarily large. It is also the primary means of defining such functions in LISP. As an example, consider the mathematical function factorial, where factorial(n) returns the product of the first n positive integers. For instance, factorial(3) is \((1 \times 2 \times 3)\), or 6; factorial(0), by definition, is 1. We can mathematically define factorial recursively as follows:

\[
\text{factorial}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times \text{factorial}(n-1) & \text{if } n > 0 
\end{cases}
\]

Our recursive definition of factorial has two distinct components. The first part of the definition (i.e., factorial(0) = 1) is known as a base-case rule, while the second part of the definition is known as a recursive rule.

Using our recursive definition, we can formally derive the value of factorial(3):
factorial(3) = 3 × factorial(2)
= 3 × (2 × factorial(1))
= 3 × (2 × (1 × factorial(0)))
= 3 × (2 × (1 × 1))
= 3 × (2 × 1)
= 3 × 2
= 6

Notice that the first three steps of the derivation of use the recursive rule from our definition. Once factorial(0) appears in our expression, we know that we’ve reached the base case and can substitute in 1 for factorial(0). Knowing the value of factorial(0), we in turn know the value of factorial(1); knowing the value of factorial(1) gives us the value of factorial(2), which in turn finally gives us the value of factorial(3).

We can define factorial recursively in LISP by following the rules of the recursive mathematical definition:

```lisp
(defun fact-r (n)
  (cond ((= n 0) 1) ; base case
        (t (* n (fact-r (- n 1)))))) ; recursive rule
```

Evaluation of the expression (fact-r 3) would follow precisely the steps outlined in the derivation of factorial(3) above. There would be three recursive invocations of fact-r corresponding to the three applications of the recursive rule. During the last of these recursive invocations (i.e., (fact-r 0)), the base case is encountered, resulting in the value 1 being returned so that the value of (fact-r 1) can be determined. The value of (fact-r 1) allows the evaluation of (fact-r 2), which in turn allows the evaluation of the original expression, (fact-r 3).

### 2.2 Execution Time of Recursive Functions

Let use examine how long it takes for the following recursive function to execute. Consider:
2.2. EXECUTION TIME OF RECURSIVE FUNCTIONS

; Function to compute b to the n’th power.
; e.g. (power 5 2) => 25
;
; Straightforward O(n) implementation.
;
(defun power (b n)
  (cond ((= n 0) ; base case
          1)
        (t ; recursion case
          (* b
             (power b (1- n))))))

To see what this function does, we trace it with the LISP interpreter:

-> (trace power)
(POWER)

-> (power 10 5)
  1> (POWER 10 5)
     2> (POWER 10 4))
     3> (POWER 10 3))
     4> (POWER 10 2))
     5> (POWER 10 1)
     6> (POWER 10 0)
        <6 (POWER 1)
        <5 (POWER 10)
        <4 (POWER 100)
        <3 (POWER 1000)
        <2 (POWER 10000)
        <1 (POWER 100000)

100000

This trace means that
we type in (power 10 5) which calls
  (POWER 10 5), which runs at level 1 and calls
    (POWER 10 4), which runs at level 2 and calls
      (POWER 10 3), which runs at level 3 and calls
        (POWER 10 2), which runs at level 4 and calls
          (POWER 10 1), which runs at level 5 and calls
            (POWER 10 0), which runs at level 6 and
              which returns 1 to level 5
              which returns 10 to level 4
              which returns 100 to level 3
              which returns 1000 to level 2
              which returns 10000 to level 1
              which returns 100000 to us

          Notice that it takes 5 calls to power to compute (power 10 5). In general, it takes \( n \) calls to power to compute (power \( b \) \( n \)), and the time to compute (power \( b \) \( n \)) is directly proportional to \( n \). Therefore we say that power is an \( O(n) \) function, where \( O(n) \) in this context means that there are two constants \( K > 0 \) and \( N \) such that the function execution time is not larger than \( K \cdot n \), for any \( n \geq N \). For power we can take \( K \) to be the time it takes for the code in power to execute at a single level, not counting the time required by recursive calls to power during that execution, and we can take \( N \) to be 1.

          Execution times are often said to be \( O(f(n)) \) for some function \( f(n) \) if \( n \). If \( f(n) \) and \( g(n) \) are two functions of \( n \), we say

          \[ g(n) \text{ is } O(f(n)) \]

          if and only if (by definition)

          \[ \text{there exist constants } K > 0, N \text{ such that } g(n) \leq K \cdot n \text{ for every } n \geq N. \]

          We can make a faster exponential function as follows:
2.2. EXECUTION TIME OF RECURSIVE FUNCTIONS

; Function to compute b to the n’th power.
;
; O(log n) implementation by repeated squaring.
;
(defun fast-power (b n)
  (cond ((= n 0) ; base case
          1)
        (evenp n) ; recursion case,
          (square ; even n
            (fast-power b (/ n 2))))
        (t ; recursion case,
          (* b (fast-power b (1- n)))))) ; odd n

(defun square (x) (* x x))

Tracing we get:
-> (trace fast-power)
(FAST-POWER)
-> (fast-power 10 15)
  1> (FAST-POWER 10 15)
     2> (FAST-POWER 10 14)
        3> (FAST-POWER 10 7)
           4> (FAST-POWER 10 6)
              5> (FAST-POWER 10 3)
                 6> (FAST-POWER 10 2)
                    7> (FAST-POWER 10 1)
                       8> (FAST-POWER 10 0)
                          <8 (FAST-POWER 1)
                             <7 (FAST-POWER 10)
                                <6 (FAST-POWER 100)
                                   <5 (FAST-POWER 1000)
                                      <4 (FAST-POWER 1000000)
                                         <3 (FAST-POWER 10000000)
                                            <2 (FAST-POWER 1000000000000000)
                                               <1 (FAST-POWER 10000000000000000000)
                                                  1000000000000000000000000000000000000000

Here we get speed for even n by going straight to \( \frac{n}{2} \) and then just squaring the result. Careful examination shows that for any n it takes at most two recursions of fast-power to divide n by
2, so the running time is at most twice the time \texttt{fast-power} code takes to execute at a single
level, times $\log_2 n$. So the execution time is $O(\log_2 n)$. But since

$$\log_2 n = (\log_2 10)(\log_{10} n)$$

any time that is $O(\log_2 n)$ is equivalently $O(\log_{10} n)$, or as we shall say, $O(\log n)$. In other words,
when you say something is $O(\log n)$, the base of the logarithm does not matter, because choosing a
different base just multiplies the logs by a constant.

2.3 Abstraction in LISP

Functions can be values of variables and arguments of other functions. Consider:

```lisp
-> (defun twice (f x) ; Apply f twice to x.
    (funcall f (funcall f x)))
TWICE
-> (twice #'1+ 3)
5
-> (twice #'cdr '(a b c d))
(C D)
```

If a symbol $s$ names a function definition, then $#' s$ is the function itself, and can be used as a
variable value and function argument.

Besides copying a function value around, you can call it by using \texttt{funcall} or \texttt{apply}. The
difference between \texttt{funcall} and \texttt{apply} is illustrated as follows:

```lisp
-> (funcall #'+ 1 2 3 4 5)
15
-> (apply #'+ '(1 2 3 4 5))
15
```

The arguments to the function called by \texttt{funcall} are the arguments to \texttt{funcall} itself, other
than the first argument, which is the function. The arguments to the function called by \texttt{apply} are
instead the elements of the second argument to \texttt{apply}, which must be a list.

Functions are means of representing procedures for doing things. Historically they have been
called \textit{procedural abstractions}, i.e. formal, hence abstract, representations of procedures.

Consider the problems:

1. Write a function \texttt{sum-integers} that computes

$$\sum_{n=\text{first}}^{n=\text{last}} n$$
2.3. ABSTRACTION IN LISP

2. Write a function `sum-squares` that computes

\[ \sum_{n=\text{first}}^{n=\text{last}} n^2 \]

with `first` and `last` as integer parameters.

3. Write a function `sum-terms` that computes

\[ \sum_{n=\text{first}}^{n=\text{last}} \text{term-fn}(n) \]

with `first` and `last` as integer parameters and `term-fn` as a function parameter.

The first two problems are easily solved by:

```lisp
(defun sum-integers (first last)
  (if (> first last)
      0
      (+ first
          (sum-integers (+ first) last))))

(defun square (x) (* x x))

(defun sum-squares (first last)
  (if (> first last)
      0
      (+ (square first)
          (sum-squares (+ first) last))))
```

When beginning students solve the third problem, they often write something like:
; Sum of (term-fn n) for an integer n running from first
; to last, inclusive.
;
; *** INCORRECT VERSION ***
;
(defun bad-sum-terms (term-fn first last)
  (if (> first last)
      0
      (+ (term-fn first)
          ; ^^^^^^^ -- ERROR, term-fn does not reference argument
          (bad-sum-terms term-fn (1+ first) last)))))

This does not work because COMMONLISP does not permit variable names to be used as the first
element of an s-expression to be evaluated, and a function argument name is a variable name.
Bad-sum-terms can be fixed by using funcall:

; Sum of (term-fn n) for an integer n running from first
; to last, inclusive.
;
; *** CORRECTED by use of "funcall" ***
;
(defun sum-terms (term-fn first last)
  (if (> first last)
      0
      (+ (funcall term-fn first)
          (sum-terms term-fn (1+ first) last)))))

Bad-sum-terms can also be fixed by using apply:

; Sum of (term-fn n) for an integer n running from first
; to last, inclusive.
;
; *** CORRECTED by use of "apply" ***

(defun apply-sum-terms (term-fn first last)
  (if (> first last)
      0
      (+ (apply term-fn (list first))
          (apply-sum-terms term-fn (1+ first) last)))))

However, it is clearly more awkward to use apply than funcall in this case because you need
an extra function call to list to prepare the arguments for apply.
Some example uses of sum-terms are:

```lisp
-> (sum-terms #'square 1 10)
385
```

```lisp
-> ; Evaluate pi as
; 8*[1/(1*3) + 1/(5*7) + 1/(9*11) + ...]
;
(defun pi-term (n)
   (/ 1.0 (* (1+ (* 4 n)) (+ 3 (* 4 n)))))))
PI-TERM
-> (defun pi-sum (howmany)
   (* 8 (sum-terms #'pi-term 0 howmany)))
PI-SUM
-> (pi-sum 8)
3.086079801123833
-> (pi-sum 200)
3.139105095248946
```

Clearly a large number of terms are required to get much accuracy. Note that for reasons of numerical accuracy, a sum like this should really be added in the other order, from the smallest terms to the largest.

Above we have used `#'` with a symbol to denote the function definition of the symbol, as defined by `defun`. We can also use `#'` with a list beginning with `lambda` that defines a function directly, as in:

```lisp
-> (mapcar #'(lambda (x) (+ x 10)) '(1 3 5))
(11 13 15)
```

Here `mapcar` applies the function to each element of a list and returns the results as a list. In this case we have defined a function that returns its single argument plus 10.

An expression of the form:

```lisp
(lambda (SYMBOL*) EXPRESSION*)
```

is called a lambda abstraction. Applying `#'` to a lambda abstraction produces a callable function.

The following is a more complex and useful example:
; Definite integral of f(x) from x = a to x = b, obtained by
; dividing the interval [a, b] into subintervals of length dx.
;
; The integral is approximated by taking the value of f(x) at the
; midpoint of each interval of length dx, multiplying by dx, and
; summing the results. That is:
;
; \[
\frac{(b-a-dx)/dx}{
\sum_{n=0}^{(integral \ f \ a \ b \ dx)} \frac{dx \ast f(a + n*dx + dx/2)}{\n}}
\]

(defun integral (f a b dx)
  (* dx
   (sum-terms #'(lambda (n)
       (funcall f (+ a
          (* dx n)
          (/ dx 2))))
      0
      (/ (- b a dx) dx))))

which can be used as follows:
-> (integral #'(lambda (x) (sqrt (- 1 (* x x)))) -1 1 0.01)
1.570968428950005

Notice that the lambda abstraction inside integral contains only one argument, n, but also uses
f, a, b, and dx, which it obtains from the environment in which it is defined. More specifically,
this environment is the set of all variables, local and global, that exist at the time the expression
#' (lambda (n) ...) is evaluated.

### 2.4 Iteration in LISP

In a previous section we defined factorial recursively in LISP by following the rules of the
recursive mathematical definition:
2.4. *ITERATION IN LISP*

; Recursive version of factorial.
(defun fact-r (n)
  (cond ((= n 0) 1) ; base case
        (t (* n (fact-r (- n 1)))))) ; recursive rule

The recursive fact-r has one problem. If we try to take the factorial of a large number, such as 10,000, we will probably get an error message saying “stack overflow”. The stack is the piece of memory which holds the arguments of functions. Each time a function is called, stack space is allocated, and each time a function returns, stack space is freed. A recursive function such as fact-r, when called with 10,000, will have 10,000 sets of arguments in simultaneous existence, and needs a big stack. LISP systems think that programs that need big stacks are likely to be erroneous, so they give you a small stack and an error message when it overflows. (They usually also give you a means of getting a larger stack if you need it.)

Since factorial(n) is defined to be the product of the first n positive integers, we can use iteration to define factorial in LISP by looping through the first n positive integers and accumulating a result during this loop:

; iterative version of factorial (using a do-loop)
(defun fact-i (n)
  (do ((temp n (- temp 1))
       (result 1 (* result temp)))
      ((= temp 0) result)))

To take another example, we calculated \( \pi \) in a previous section using a function called sum-terms. Because the function was defined recursively, we could not sum very many terms in the calculation of \( \pi \), least the stack overflow, and therefore we got poor accuracy. We can get better accuracy with an iterative version of sum-terms:
; Sum of (term-fn n) for an integer n running from first to last, inclusive.

; Iterative version.

(defun sum-terms-i (term-fn first last)
  (do ((n first (1+ n))
       (ans 0 (+ ans (funcall term-fn n)))
       ((> n last) ans)))

; Evaluate pi as 8*[1/(1*3) + 1/(5*7) + 1/(9*11) + ...]

(defun pi-term (n) (/ 1.0 (* (1+ (* 4 n)) (+ 3 (* 4 n)))))
(defun pi-sum (howmany)
  (* 8 (sum-terms-i #'pi-term 0 howmany)))

which we may run as long as we want:

-> (pi-sum 100)
  3.1364218870299
-> (pi-sum 1000)
  3.141093153121444
-> (pi-sum 10000)
  3.14154265858932

How do our iterative and recursive versions of factorial compare? Actually, do can be implemented as a macro that produces a particular type of recursive function called a tail-recursive function. If this is done, the iterative version of factorial is equivalent to:

; iterative version of factorial (using a macro expansion of do)
(defun fact-i (n)
  (labels ((g0001 (temp result)
            (cond
             ((= temp 0) result)
             (t (g0001 (- temp 1) (* result temp)))))
    (g0001 n 1)))

Here g0001 is a symbol that was generated when the do macro expanded, and is the name of a recursive function defined by labels with arguments temp and result. Labels is just like flet, but permits the functions defined to call themselves recursively.

The function g0001 contains a tail-recursive function call to itself. A function call is said to be tail recursive if it calls the function that contains it, is the last action to be executed by that
function, and produces the resulting value of that function.

Tail-recursive function calls are implemented by LISP compilers using the following trick. Instead of calling itself, the function just takes its new argument values, sets its current arguments equal to these new values, and jumps back to the beginning of the function.

In other words, a tail-recursive function call is translated into an iterative loop, and the function never actually calls itself recursively.

Why is tail-recursion important. Well, its a matter of stack size. When a function calls itself tail-recursively, no extra argument space is needed. Tail-recursive calls are also faster than normal recursive calls because they do not need to allocate and deallocate stack memory for arguments.

A function whose only recursion is tail-recursion can get along with a very small stack. Such functions are called tail-recursive functions.

Notice that in order to make \texttt{g0001} tail-recursive, we had to introduce a second argument to hold the result.

(Our \texttt{fact-r} is not tail-recursive, since the last operator it evaluates in the recursion case is \texttt{*} in the expression \texttt{(* n (fact \ (- n 1))}). Many LISP compilers are smart enough to translate this kind of function into a tail recursive function automatically by using the fact that \texttt{*} is a commutative, associate operator with a unit element, 1, and only one of its arguments is a recursive call. In the process of this translation the order of multiplications is reversed. Also, your LISP might be so smart as to turn \texttt{fact-r} into a tail recursive function by default, in which case you will not be able to generate a stack overflow error by calling it with a large number.)

Recursive functions are often natural solutions to problems, while iterative or tail-recursive solutions often require an extra variable or argument to hold the result, and are more complex.

The general recursive approach is to first look for base cases for which the solution of the problem is known. The next step is to consider the general case (i.e., all situations other than the base cases) and figure out a way to simplify the problem in the general case by expressing the problem in terms of some simpler case. The goal is to use a simplification that ultimately simplifies any problem to the base cases.

Since lists are the basic data structures within LISP, many of the functions that you’ll want to write in LISP will serve to manipulate the expressions within a given list. Recursion is often a helpful tool in performing such manipulations. As an example, consider a function \texttt{count-atoms} that determines the number of non-nil atoms that appear in a list. For example, if \texttt{count-atoms} is passed \texttt{((a (b)) a b (c))} as an argument, it should return the value \texttt{5}. This function works on all of the levels in a list, not just the top-level. Here is a recursive implementation of \texttt{count-atoms}, called \texttt{count-atoms-r}:
; a recursive version of count-atoms having two base cases,
; returning 0 if lst is empty and 1 if lst is an atom.
; If lst is a list, there is a recursive call on both its car
; and cdr in order to handle atoms on all levels of the list.
(defun count-atoms-r (lst)
  (cond ((null lst) 0)
        ((atom lst) 1)
        (t (+ (count-atoms-r (car lst))
               (count-atoms-r (cdr lst))))))

Let's try an iterative implementation of count-atoms. To see what would be involved, think
back to the dotted-pair box notation for LISP expressions. For count-atoms to work correctly
with multi-leveled lists, it's necessary to remember a list of all the dotted-pairs whose cdrs you
have not looked at yet while you are looking at the car of some dotted-pair. This can be done in a
fairly straightforward manner by the following code:

; an iterative version of count-atoms that maintains a to-do list
; of all the cons cells whose cdrs have not yet been looked at.
(defun count-atoms-i (lst)
  (do ((to-do nil)
       (current lst)
       (result 0))
      ((and (null to-do) (null current)) result)
    (cond
      ((atom current)
        (if current (setf result (+ 1 result)))
        (setf current (cdr (car to-do)))
        (setf to-do (cdr to-do))))
    (t
      (setf to-do (cons current to-do))
      (setf current (car current))))))

In the recursive version of count-atoms the list of cons cells whose cdrs had not yet been
visited is maintained in the stack. In the iterative version above, it is maintained as the LISP list
to-do created by calls to cons. It turns out that it is faster to maintain lists in the stack, because
allocating and deallocating stack memory is faster than allocating and garbage collecting cons
cells.

We can change the above code so that the to-do list is in the stack but iteration is still used
to move along a level without adding to the stack depth. The resulting composite count-atoms
is our most efficient version, and handles arbitrarily long lists without stack overflow, but can still
2.4. **ITERATION IN LISP**

suffer stack overflow if given arbitrarily deep list structures:

```lisp
(defun count-atoms (lst)
  (cond ((null lst) 0)
        ((atom lst) 1)
        (t (do ((templst lst (cdr templst))
               (result 0 (+ result
                      (count-atoms (car templst))))
               ((atom templst)
                (if templst (+ 1 result) result))))))
```

The `count-atoms` function is often called *doubly recursive* because it combines (e.g. adds) the results of two calls to itself. The `fact` function is called *singly recursive* because it requires only one call to itself to get information to combine into its result. As we have seen, iteration can eliminate single recursion, or half of double recursion, at the cost of adding an extra variable to hold the result. Recursion is often the more natural approach, because it uses one less variable, but iteration is often the more efficient approach, because it eliminates stack memory allocations and deallocations. There are also some situations in which recursion is unnatural, such as input or output loops, or loops with a fixed number of iterations (say 17).

The `do` statement is not the only way to specify iteration in LISP. The following version of `count-atoms` is very similar to the previous version:

```lisp
(defun count-atoms-map (lst)
  (cond ((null lst) 0)
        ((atom lst) 1)
        (t (apply '+ (mapcar 'count-atoms-map lst)))))
```

This last version is perhaps one of the easiest to read. Some LISP compilers are smart enough to turn such `apply/mapcar` constructs into do-loops. Otherwise `apply/mapcar` is an inefficient way to express a do-loop since `mapcar` builds an intermediate and unnecessary list for `apply` to work on. In this case the other problem is that this last function is no longer truly equivalent to the original doubly recursive `count-atoms-r`, in that dotted lists are no longer handled.

The moral of the story is that there are a lot of ways to go about solving a single programming problem in LISP. In addition to a wealth of helpful LISP functions, you have at your disposal the techniques of recursion and iteration. Try to make the best possible use of your resources in order to write code that is simple and readable, yet not unbearably inefficient.
CHAPTER 2. A TOUR OF LISP

2.5 Symbolic Differentiation

Let us apply what we have learned to writing a serious program. In doing so, we will use a software engineering methodology that has been developed to make the writing of programs more disciplined. This scheme involves writing the program four times: first as a specification, second as a preliminary design, third as a detailed design, and fourth as code. Each time we write the program we will add more details.

2.5.1 Specification

Write a function \texttt{deriv} that will compute the symbolic derivative of an algebraic expression with respect to a variable. For example:

\[
\texttt{-> (deriv '(+ (* 5 \textit{x}) (* 2 (expt \textit{x} 4))) 'x)} \\
(+ 5 (* 8 (expt \textit{x} 3)))
\]

The first argument is the algebraic expression, and the second is the variable with respect to which the derivative is being taken.

The algebraic binary operators +, -, *, /, and \texttt{expt} should be handled, but the power argument in \texttt{expt} may be required to be a number. The result should be simplified at least to the extent that numbers are combined arithmetically when possible: e.g. 11 should be returned and not (+ 5 6).

Notes on Specification: A specification is a contract between the user of the program and the implementer of the program stating what the program is to do. Any detail important to the user must be included, but any detail the implementer is free to choose must be excluded.
2.5. SYMBOLIC DIFFERENTIATION

2.5.2 Preliminary Design without Simplification

The program will recursively apply the following rules, where \( v \) is the variable being differentiated by, \( x \) is any variable, \( n \) is any number, and \( \alpha, \beta \) are algebraic expressions:

\[
\frac{dv}{dv} = 0 \text{ for any constant } c
\]
\[
\frac{dx}{dv} = \begin{cases} 
1 & \text{if } x = v \\
0 & \text{if } x \neq v 
\end{cases}
\]
\[
\frac{d}{dv}(\alpha + \beta) = \frac{d\alpha}{dv} + \frac{d\beta}{dv}
\]
\[
\frac{d}{dv}(\alpha \cdot \beta) = \frac{d\alpha}{dv} \cdot \beta + \alpha \cdot \frac{d\beta}{dv}
\]
\[
\frac{d}{dv}\left(\frac{\alpha}{\beta}\right) = \frac{\beta \cdot \frac{d\alpha}{dv} - \alpha \cdot \frac{d\beta}{dv}}{\beta^2}
\]
\[
\frac{d}{dv}(\alpha^n) = n \cdot \alpha^{n-1} \cdot \frac{d\alpha}{dv}
\]

The deriv function will select one of the above rules based on the input expression, and thus will have the form:

```
derv (exp var):
    is exp constant? => apply constant rule
    is exp = var? => apply variable rule
    is exp a sum? => apply sum rule
    is exp a difference? => apply difference rule
    is exp a product? => apply product rule
    is exp a quotient? => apply quotient rule
    is exp an exponential? => apply exponential rule
```

There will be separate functions to make expressions: make-sum, make-diff, make-product, make-quotient, and make-power. The simplification rules will be embedded in these functions, so the simplification rules can be added later or changed just by changing these functions.

Notes on Preliminary Design: A preliminary design is a high level design of the program. It specifies major algorithms, and the organization of major data structures and major program structures.

Notes on Staged Modular Development: Often it is a good idea to develop a program in stages, modularizing the program so the parts developed in each stage fit together well. We are doing this here by deferring development of simplification, while providing an interface for the simplification code to be included later.
2.5.3  Detailed Design

The following is the detailed design, which is essentially just a program skeleton:

;  *******************************************************
;  Function to take symbolic derivatives.
;  *******************************************************

;  (deriv e v) returns the derivative de/dv.
;
;  The expression e is of the form:
;
;  expr ::= number | variable |
;           (+ expr expr) | (- expr expr) |
;           (* expr expr) | (/ expr expr) |
;           (expt expr number)
;
(defun deriv (exp var)
  (cond
    ((constant? exp) 0)
    ((variable? exp)
      (if (same-variable? exp var) 1 0))
    ((sum? exp)
      {calculate sum rule using:
        make-sum, first-addend, second-addend})
    ((diff? exp)
      {calculate difference rule using:
        make-diff, diminuend, subtrahend}))
2.5. SYMBOLIC DIFFERENTIATION

((product? exp)
 {calculate product rule using:
  make-sum, make-product,
  first-factor,
  second-factor})
((quotient? exp)
 {calculate quotient rule using:
  make-quotient, make-diff,
  make-product, make-power,
  numerator, denominator})
((power? exp)
 {calculate power rule using:
  make-product, make-power,
  base, exponent})))

; ******************************************************
; Functions defining the representation of expressions
; by lists
; ******************************************************

(defun constant? (x) {})
(defun variable? (x) {})
(defun same-variable? {})
(defun sum? (x) {})
(defun first-addend (x) {})
(defun second-addend (x) {})
(defun make-sum (x1 x2) {})
(defun diff? (x) {})
(defun diminuend (x) {})
(defun subtrahend (x) {})
(defun make-diff (x1 x2) {})

(defun product? (x) {})
(defun first-factor (x) {})
(defun second-factor (x) {})
(defun make-product (x1 x2) {})

(defun quotient? (x) {})
(defun numer (x) {})
(defun denomin (x) {})
(defun make-quotient (x1 x2) {})

(defun power? (x) {})
(defun base (x) {})
(defun exponent (x) {})
(defun make-power (x1 x2) {})

Here code that has not been written yet is represented by a verbal description in curly brackets.

Notes on Detailed Design: A detailed design is a partial coding of the program whose purpose is to determine what functions are to be written and what they do, and what data structures are to be used. Any detail not relevant to this purpose should not be given.

2.5.4 Code without Simplification

The code for a basic version of the program that does not include algebraic simplification of expressions is:
2.5. SYMBOLIC DIFFERENTIATION

; Program to take symbolic derivatives.
;
; File: deriv.lsp
; Author: course (lib51@husc)
; Version: 1
;
; *******************************************************
; Function to take symbolic derivatives.
; *******************************************************

; (deriv e v) returns the derivative de/dv.
;
; The expression e is of the form:
;
; expr ::= number | variable |
; (+ expr expr) | (- expr expr) |
; (* expr expr) | (/ expr expr) |
; (expt expr number)
;
(defun deriv (exp var)
  (cond
    ((constant? exp) 0)
    ((variable? exp)
      (if (same-variable? exp var) 1 0)))
((sum? exp)
 (make-sum
  (deriv (first-addend exp) var)
  (deriv (second-addend exp) var))
)((diff? exp)
 (make-diff (deriv (diminuend exp) var)
            (deriv (subtrahend exp) var))
)((product? exp)
 (make-sum
  (make-product
   (first-factor exp)
   (deriv (second-factor exp) var))
  (make-product
   (deriv (first-factor exp) var)
   (second-factor exp)))
)((quotient? exp)
 (make-quotient
  (make-diff
   (make-product
    (denominator exp)
    (deriv (numerator exp) var))
    (make-product
     (numerator exp)
     (deriv (denominator exp) var)))
   (make-power (denominator exp) 2))
)((power? exp)
 (make-product
  (exponent exp)
  (make-product
   (make-power (base exp)
               (- (exponent exp) 1))
   (deriv (base exp) var))))
2.5. SYMBOLIC DIFFERENTIATION

; Functions defining the representation of expressions by lists

(defun constant? (x) (numberp x))

(defun variable? (x) (symbolp x))

(defun same-variable? (x1 x2)
    (and (variable? x1) (variable? x2) (eq x1 x2)))

(defun sum? (x) (and (listp x) (eq (car x) '+)))
(defun first-addend (x) (cadr x))
(defun second-addend (x) (caddr x))
(defun make-sum (x1 x2) (list '+ x1 x2))

(defun diff? (x) (and (listp x) (eq (car x) '-)))
(defun diminuend (x) (cadr x))
(defun subtrahend (x) (caddr x))
(defun make-diff (x1 x2) (list '-' x1 x2))

(defun product? (x) (and (listp x) (eq (car x) '*)))
(defun first-factor (x) (cadr x))
(defun second-factor (x) (caddr x))
(defun make-product (x1 x2) (list '*' x1 x2))

(defun quotient? (x) (and (listp x) (eq (car x) '/)))
(defun numer (x) (cadr x))
(defun denomin (x) (caddr x))
(defun make-quotient (x1 x2) (list '/' x1 x2))

(defun power? (x) (and (listp x) (eq (car x) 'expt)))
(defun base (x) (cadr x))
(defun exponent (x) (caddr x))
(defun make-power (x1 x2) (list 'expt x1 x2))
The following is a simple test of this program:

```lisp
-> (load '|deriv.l|)
T
-> (deriv '(+ x (* 8 (+ x y))) 'x)
(+ 1 (+ (* 8 (+ 1 0)) (* 0 (+ X Y))))
-> (deriv '(+ (* a (expt y 2)) (* b y)) 'y)
(+ (+ (* A (* 2 (* (EXPT Y 1) 1))) (* 0 (EXPT Y 2)))
   (+ (* B 1) (* 0 Y)))
```

2.5.5 Preliminary Design of Simplification

Now that we have a working program, we will add simplification.

Simplification will be done by applying the following rules when creating arithmetic expressions:

- If all arguments in an expression are numbers, the expression will be reduced to a number.
- If a sum or product has a numeric term, it will be placed as the first argument.
- Addition of 0, multiplication by 1, and exponentiation by the power 1 will be eliminated.
- Multiplication by 0 will be reduced to a 0 result. Exponentiation by the power 0 will be reduced to the result 1.
- When two sums are summed, if both have numeric terms these will be combined. Ditto when two products are multiplied.
- Subtraction will be rewritten as summation and multiplication by -1. Division will be rewritten as multiplication and exponentiation to the power -1.

2.5.6 Final Code

In order to simplify expressions, all we must do is revise the `make-sum`, `make-diff`, `make-product`, `make-quotient`, and `make-power` routines:
2.5. SYMBOLIC DIFFERENTIATION

; Functions that simplify algebraic expressions.
;
; File: simplify.lsp
; Author: course (lib51@husc)
; Version: 1
;
; Load this file after loading deriv.l
;
; ***********************************************
; Functions to make algebraic expressions with modest
; algebraic simplification
; ***********************************************

; When we make sums we put the number part first.
(defun make-sum (x1 x2)
  (cond
    ((and (numberp x1) (zerop x1))
     x2)
    ((and (numberp x1) (numberp x2))
     (+ x1 x2))
    ((numberp x2) (make-sum x2 x1))
    ((and (numberp x1)
         (sum? x2)
         (numberp (first-addend x2)))
     (make-sum (+ x1 (first-addend x2))
               (second-addend x2)))
    ((and (sum? x1)
         (numberp (first-addend x1))
         (sum? x2)
         (numberp (first-addend x2)))
     (make-sum (+ (first-addend x1) (first-addend x2))
               (make-sum (second-addend x1)
                         (second-addend x2))))
    (t
     (list '+ x1 x2))))
; Differences are rewritten as sums and products
(defun make-diff (x1 x2)
  (make-sum x1 (make-product -1 x2)))

; When we make products we put the number part first,
; and we apply the sum-product distribution rule.
(defun make-product (x1 x2)
  (cond
    ((and (numberp x1) (zerop x1)) 0)
    ((and (numberp x1) (= x1 1)) x2)
    ((and (numberp x1) (numberp x2)) (* x1 x2))
    ((numberp x2) (make-product x2 x1))
    ((sum? x1)
      (make-sum (make-product (first-addend x1) x2)
                (make-product (second-addend x1) x2)))
    ((sum? x2)
      (make-sum (make-product x1 (first-addend x2))
                (make-product x2 (second-addend x2))))
    ((and (numberp x1)
          (product? x2))
      (make-product (* x1 (first-factor x2))
                    (second-factor x2)))
    ((and (product? x1)
          (numberp (first-factor x1))
          (product? x2)
          (numberp (first-factor x2))
          (make-product (* (first-factor x1) (first-factor x2))
                        (make-product (second-factor x1) (second-factor x2))))))
    (t
      (list '* x1 x2))))
2.5. SYMBOLIC DIFFERENTIATION

(defun make-quotient (x1 x2)
 (make-product x1 (make-expt x2 -1)))

(defun make-power (x1 x2)
 (cond
   ((and (numberp x1) (numberp x2))
    (expt x1 x2))
   ((and (numberp x2) (zerop x2))
    1)
   ((and (numberp x2) (= x2 1))
    x1)
   (t
    (list 'expt x1 x2))))

We can rerun the test of deriv.l with these algebraic simplifiers:
-> (load '|deriv.l|)
T
-> (load '|simplify.l|)
T
-> (deriv '(+ x (* 8 (+ x y))) 'x)
9
-> (deriv '(+ (* a (expt y 2)) (* b y)) 'y)
(+ (* A (* 2 Y)) B)

2.5.7 Rationale for Software Engineering Methodology

Why should we bother writing a program specification, then a preliminary design, then a detailed design, and then the final code?

The main problem we face when we are asked to write a program is that we are ignorant of a precise algorithm, an exact data organization, and an exact program organization that will get good results. There is a major problem with understanding all the details and getting all the small parts to fit together smoothly.

Usually the only way to solve this problem is to write the program more than once. With each draft we discover details that do not fit together, and adjust the program as a whole to fix it.

But there is no reason why we cannot begin by writing the program in a very high level language, like English and mathematics, and suppress all details that are unnecessary to figuring out whether we have the right algorithm. This is what the preliminary design is. Because we sup-
press many details, we can write the preliminary design much faster than we could write the whole program.

Similarly we can write a version of the program that indicates just the main parts of the program, the flow of control between these parts, and the interfaces between the parts, while suppressing all details not needed to figuring out whether our program organization is correct. This is what the detailed design is. Because we suppress many details, we can write the detailed design faster than we could write the whole program.

So by writing a preliminary design, a detailed design, and finally the code, we get the effect of writing the program three times, but at a much reduced cost.

When can you profitably skip writing all these different versions of the program? The answer is: when you already know what you would learn from the version. If you have the algorithms, control flow, and data structures down pat, there may be little reason to write a preliminary design, for example.

The assignments in this course give you a specification, of course, because specifications are just problem statements. The assignments generally also give you most of the preliminary design, because we are teaching algorithms that took computer scientists some time to discover, and it is inappropriate to ask you to rediscover them from scratch in one week. Therefore, for the most part you are asked to do just detailed design, code, and test. There is little reason to skip doing detailed design, as it directly becomes part of the final code, and needs to be done anyway.

Software engineers use the four step methodology for another reason which has little to do with course assignments. When a single large program is written by many people, it is much easier for all these people to deal with shorter descriptions of the program, such as preliminary designs. Therefore the people will write and read specifications, preliminary designs, and detailed designs as a group, but will typically read little code but their own.

Lastly, we should mention that there is a totally different software engineering method that may be of use in doing course assignments. Software engineers may run experiments in order to learn about something specific. For example, if you did not understand the various LISP equality checking functions, \texttt{eq}, \texttt{equal}, \texttt{eql}, and \texttt{=}, very well, you might experiment with them while doing detailed design. The fact that you needed to understand these functions might not occur to you until you begin doing detailed design, so the need for experimenting can arise at any time.

Often these experiments take the form of writing a simplified version of part of the final program. This is called prototyping. Sometimes these experiments involve measuring the performance of operations (e.g. \texttt{cons} or \texttt{car}) or algorithms. This is called benchmarking. But the idea of running specific experiments to fill in specific gaps in one’s information is more general than either prototyping or benchmarking.
2.6 Memoizing

Consider the following simple recursive implementation of the fibonacci function (which is interesting because, among other things, it controls the spacing of leaves on plants):

```lisp
(defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2)))))
)
```

This code is called purely functional because it does not use any form of "memory", such as variables. Its all just a matter of evaluating functions.

How many calls to fib does it take, counting recursive calls, to compute \( (\text{fib } n) \)? Let this number be called \( C_n \). Then

\[
C_n = \begin{cases} 
1 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
1 + C_{n-1} + C_{n-2} & \text{if } n > 1 
\end{cases}
\]

If we chose \( \beta \) and \( \gamma \) so that

\[
1 = \beta^{-1} + \beta^{-2}, \beta > 1
\]

and

\[
\beta < \gamma
\]

then by using mathematical induction it is possible to prove

\[
C_n \geq \beta^{n-2} \text{ for } n > 0
\]

and

\[
C_n < \gamma^n \text{ for large enough } n
\]

We then solve for \( \beta \) and find

\[
\beta = \frac{1}{2} + \sqrt{\frac{3}{4}} = 1.366025388240814
\]
which permits us to take
\[ \gamma = 1.367 \]
which is what we did above when we announce that \((\text{fib } n)\) executed in time \(O(1.367^n)\) time.

Using the lower bound on \(C_n\) we find that
\[ C_{100} \geq \beta^{98} > 18,800,000,000,000 \]
so it would take even a very fast computer many days to compute \((\text{fib } 100)\).

There must be a better way. Suppose that, whenever we evaluated \((\text{fib } n)\) for some \(n\), the \texttt{fib} function looked in a memory to see if it had already done this computation before, and if yes, simply read the result from the memory. If no, then \texttt{fib} would make the recursive calls as before.

This way of using memory is called \textit{memoizing}. Let us call our new memoizing fibonacci function \texttt{memo-fib} and ask how many calls to \texttt{memo-fib} does it take to compute \((\text{memo-fib } n)\)? If we call this number \(C'_n\) and assume that \((\text{memo-fib } n)\) evaluates \((\text{memo-fib } (- n 1))\) before evaluating \((\text{memo-fib } (- n 2))\) then:

\[
C'_n = \begin{cases} 
1 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
1 + C'_{n-1} + 1 & \text{if } n > 1 
\end{cases}
\]
as the call to evaluate \((\text{memo-fib } (- n 2))\) will find the result already computed and make no further calls. Therefore:
\[ C'_n = 2n - 1 \text{ if } n > 0 \]
and \texttt{memo-fib} takes only \(O(n)\) time to execute.

Now let us write \texttt{memo-fib}. We could just write it directly, but LISP allows us to build a general tool that simplifies the definition of any memoizing function. Our particular tool for this purpose will be a function named \texttt{memoize} with the following detailed design:
2.6. MEMOIZING

; A memoizer. Takes an argument, a memory name (a symbol),
; and a function (i.e. a lambda-expression). Checks the
; argument to see if the value of the function on the
; argument has already been computed, and if yes, returns
; that value. If no, applies the function to the argument,
; memorizes the result, and returns the result.

; The memory is a global variable with given memory name
; that stores a list of the form:
; ((arg1 value1) (arg2 value2) ... )

(defun memoize (arg name fn)
  (if {memory not initialized} {initialize memory})
  (let ((pair {(arg value) pair from memory
              or nil if none}))
    (cond
      ((null pair)
        {compute pair from arg and fn}
        {store pair in memory}))
      {return value from pair}))

Given this memoizer we can define memo-fib by:

; Memo-ized version of fibonacci function
; -- runs in time O(n)
(defun memo-fib (n)
  (memoize n 'memo-fib-values
    #'(lambda (n)
        (cond ((= n 0) 0)
              ((= n 1) 1)
              (t (+ (memo-fib (- n 1))
                  (memo-fib (- n 2)))))))

The code for memoize is as follows:
A memoizer. Takes an argument, a memory name (a symbol), and a function (i.e. a lambda-expression). Checks the named memory to see if the value of the function on the argument has already been computed, and if yes, returns that value. If no, applies the function to the argument, memorizes the result, and returns the result.

The memory is a global variable with given memory name that stores a list of the form:

$$((\text{arg}_1 \ \text{value}_1) \ (\text{arg}_2 \ \text{value}_2) \ \ldots)$$

(defun memoize (arg name fn)
  ; Initialize memory
  (if (not (boundp name)) (set name nil))

  (let ((pair (assoc arg (eval name))))
    (cond
     ((null pair)
      (setf pair (list arg (funcall fn arg)))
      (set name (cons pair (eval name)))))
    (cadr pair)))

A test of the above is:

```
-> (memo-fib 100)
354224848179261915075
-> (memo-fib 200)
280571172992510140037611932413038677189525
```

### 2.7 Lazy Evaluation

Suppose we want to define a list of all natural numbers, from 0 onward. This is an infinite list, so we will have to be intelligent about it. We shall call such an unbounded list a *stream*.

The basic idea is to represent a stream by a cons-cell whose car is the next element of the stream, and whose cdr is a function of no arguments that computes the “rest” of the stream. An empty stream is represented by *nil*. The following code can handle this task:
2.7. LAZY EVALUATION

; Prototype definition of the stream of natural numbers.

; Function to return a stream of integers beginning with n.
;
(defun integers-starting-from (n)
  (cons n #'(lambda () (integers-starting-from (1+ n))))))

(defvar naturals (integers-starting-from 0))

(defun car-stream (stream) (car stream))

(defun cdr-stream (stream) (funcall (cdr stream)))

(defun null-stream (stream) (null stream))

We can try this out:

-> (car-stream naturals)
0
-> (car-stream (cdr-stream naturals))
1
-> (car-stream (cdr-stream (cdr-stream naturals))))
2
-> (cdr-stream naturals)
(1 LAMBDA-CLOSURE ((N 1)) NIL
  ((INTEGERS-STARTING-FROM BLOCK #<@00154718>) NIL
   (INTEGERS-STARTING-FROM (1+ N)))))

The list in the cdr of the last evaluation is actually a lambda abstraction that is a list beginning with lambda-closure instead of the usual lambda, and this represents the results of evaluating the lambda abstraction (the #'(lambda ...) expression) in integers-starting-from.

Car-stream and cdr-stream seem to work fairly well, as does the data structure we have chosen to represent streams. But how can we define cons-stream? We would like to write:

; Function to return a stream of integers beginning with n.
;
(defun integers-starting-from (n)
  (cons-stream n (integers-starting-from (1+ n))))

The problem is that the second argument to cons-stream should not be evaluated when cons-
CHAPTER 2. A TOUR OF LISP

stream is called: its evaluation should actually be delayed until cdr-stream is called on the stream we are consing. Thus cons-stream cannot be a function.

We solve this problem by writing cons-stream as a macro:

; Macro to cons a stream with a given first element and a given expression whose evaluation is to be delayed until cdr-stream is called.

; (defmacro cons-stream (car-element cdr-expr)
  (list 'cons car-element
        (list 'function (list 'lambda () cdr-expr))))

which when applied gives the following:

-> (macroexpand '(cons-stream x y))
(CONS X #'(LAMBDA () Y))

-> (macroexpand '(cons-stream n (integers-starting-from (1+ n))))
(CONS N #'(LAMBDA () (INTEGERS-STARTING-FROM (1+ N)))))

Now that we have a complete definition of the basic stream data type, we can add some higher level functions:

; nth-stream returns the (0-based) nth element of a stream.

(defun nth-stream (n stream)
  (if (zerop n)
      (car-stream stream)
      (nth-stream (1- n) (cdr-stream stream))))

; unfold takes a stream and returns the corresponding list
; --- forces all delayed evaluations.

(defun unfold-stream (stream)
  (if (null-stream stream)
      nil
      (cons (car-stream stream)
            (unfold-stream (cdr-stream stream)))))

Next we can give various applications that can use streams. First we consider a finite stream that can be unfolded.
2.7. LAZY EVALUATION

; Enumerate-interval returns a stream of successive integers.
; It is used as if working with lists, but the
; elements will be generated only as needed.

(defun enumerate-interval (low high)
  (if (> low high)
      nil
      (cons-stream low (enumerate-interval (1+ low) high))))

-> (setf seq1-10 (enumerate-interval 1 10))
(1 LAMBDA-CLOSURE ...)
-> (unfold-stream seq1-10)
(1 2 3 4 5 6 7 8 9 10)

We can make streams from other streams. A standard operation is to filter elements of a stream,
accepting only those that satisfy a given predicate.

; Filter is an operation on streams. Its output stream
; contains only elements of the input stream that satisfy
; the predicate pred.

(defun filter (pred stream)
  (cond ((null-stream stream) nil)
        ((funcall pred (car-stream stream))
         (cons-stream (car-stream stream)
                      (filter pred (cdr-stream stream))))
        (t (filter pred (cdr-stream stream))))

; No-sevens is the stream of all natural numbers that are
; NOT divisible by 7.

(defun divisible? (x y)
  (zerop (mod x y)))

; (setf no-sevens
    (filter #'(lambda (x) (not (divisible? x 7)))
           naturals))
-> (nth-stream 100 no-sevens)
117
-> (nth-stream 200 no-sevens)
234

Finally, we can program fibonacci in yet another way. Build a stream such that an element is the sum of the previous two, given two initial values; now put the stream that starts at (0 1) in variable fib, and then get any element you want.

(defun fibgen (a b)
  (cons-stream a (fibgen b (+ a b))))

(setf fibs (fibgen 0 1))

-> (nth-stream 10 fibs)
55

We leave it to the reader to determine whether the execution time of fibgen is $O(1.367^n)$ or $O(n)$.

Delayed evaluation can be used more generally than just in streams. We can, for example, write a delay macro that can produce a delayed expression which can be computed later by means of a force function:
2.8. DESTRUCTIVE FUNCTIONS IN LISP

; Delay takes an expression and returns a closure having
; the environment in which the expression was going to be
; evaluated, and a body that is the expression we want
; to evaluate later.
;
(defmacro delay (expression)
  (list 'function (list 'lambda () expression)))

; Force may be applied to the result of delay to evaluate
; the expression delayed.

(defun force (delayed-object)
  (funcall delayed-object))

->(macroexpand '(delay (+ x 8)))
#'(LAMBDA () (+ X 8))
->(setf d (let ((x 5)) (delay (+ x 8))))
(LAMBDA-CLOSURE ((X 5)) () () () (+ X 8))
->(force d)
13

When the computer is confronted with an expression, the computer may chose to evaluate
it and return the result, or the computer may create a closure containing the expression and its
environment, and return that, allowing the expression to be evaluated later. The first choice is
called *eager evaluation* of the expression, and the second choice is called *lazy evaluation*.

2.8 Destructive Functions in LISP

In LISP a destructive function is one that changes a value which is not normally viewed as change-
able. Any function that changes the car or cdr of a cons cell is a destructive function, as all variables
pointing at that cons cell change their values. There are no other destructive functions in the sub-
set of COMMONLISP used in this course, as components of symbols other than their names are
meant to be changed, and there are no functions in COMMONLISP to change names of symbols
or components of numbers.

The basic destructive functions are *rplaca* and *rplacd*, and macro expressions such as:

(setf (car x) y)
(setf (cdr x) y)
Other commonly used functions are `nreverse`, `nconc`, and `sort`.

The function `nconc`, for instance, is just like its non-destructive counterpart `append` except that it actually alters the pointers of its arguments. Consider the following sequence of evaluated s-expressions:

```
-> (defvar x '(a b))
X
-> (defvar y '(c d))
Y
-> (append x y)
(A B C D)
-> x
(A B)
-> (nconc x y)
(A B C D)
-> x
(A B C D)
```

The original value of `x` is unaffected by the `append` statement but is changed (“destroyed”) by the `nconc` statement. As a non-destructive function, `append` first makes a copy of `x` (i.e., it actually constructs two new dotted-pair cells) so that the old value of `x` remains intact. Using this copy, `append` then alters the last cdr pointer in the structure so that it points to `y`’s structure. Meanwhile, `nconc` performs this second step without first making a copy of `x`:
The `sort` function is one destructive function that is not easy to avoid using. It sorts a list by changing the cdrs of the list’s cons cells, and can be made into a non-destructive function by making a copy of its input list before sorting via the `copy-list` function, as in:

```lisp
-> (defvar x '(3 7 2 1 5)) ; x points at (3 ...)  
X
-> (sort (copy-list x) #'<)
(1 2 3 5 7)
->x ; would equal (3 5 7) were it not for copy-list
(3 7 2 1 5)
```

By using LISP’s destructive functions, it’s actually possible to create self-referential expressions. As an example, we can bind the value `(a b)` to the symbol `x`. Evaluation of the expression `(rplacd (cdr x) x)` would then cause `x`’s value to be self-referential:
The `rplacd` statement changes the second cdr pointer within x’s structure so that it points back to the beginning of x. If you try to do this with the LISP interpreter, it will loop indefinitely trying to print the value of x.

Not surprisingly, COMMONLISP interpreters will infinitely loop when trying to output the expression represented by this cyclical structure. You can make them look for circular data and print it appropriately by setting the `*print-circle*` global variable to true:

```
-> (setf *print-circle* t) ; permits printing circular lists
T
-> (defvar x '(a b))
X
-> (rplacd (cdr x) x)
#0=(A B . #0#)
```

Normally `*print-circle*` is set false because it is very computationally expensive to detect that data structures are circular.

The most important destructive function in LISP is probably `nreverse`. This is because it is often easy and efficient to create some list in reverse order, and awkward or inefficient to create it in normal order. Given a newly created list in reverse order, `nreverse` can reverse it without the overhead of allocating any new cons-cells. Since the list was just created, it is typically not visible to anyone but its creator, and modifying the list destructively is not a problem.

## 2.9 Lambda Calculus and Pure LISP

LISP is based on the lambda calculus (an invention of Alonzo Church), so it is appropriate to define what the lambda calculus is. This information is more useful to understanding past history and future trends in computer science, than it is in writing programs. The contents of this section are not used elsewhere in this course.

There are many inessentially different varieties of lambda calculus. We will present a variety that uses LISP notation, and which we call Pure LISP (our dialect of Pure LISP is purely our own, and you will not find it in other books).

Pure LISP consists of a set of constants, a set of builtin functions, a set of variables, a set of formulas, and some reduction rules. We can define it in part with the grammar in Figure 2.1. Examples defined by this grammar are in Figure 2.2.

A central concept in lambda calculus is abstraction, with includes the `lambda` from which the calculus gets its name.

The rest of this section is not as precise as most of this book. We do not want to spend the time in this course to be precise in these matters, but think that a brief introduction to them may enlighten the reader nonetheless.
### Figure 2.1: Grammar of Pure LISP

<table>
<thead>
<tr>
<th>Production</th>
<th>Right-Hand Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT</td>
<td>NUMBER</td>
</tr>
<tr>
<td>BUILTIN-FUNCTION</td>
<td>SYMBOL</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>SYMBOL</td>
</tr>
<tr>
<td>ABSTRACTION</td>
<td>(lambda (VARIABLE*) FORMULA)</td>
</tr>
<tr>
<td>FUNCTION-SPECIFIER</td>
<td>BUILTIN-FUNCTION</td>
</tr>
<tr>
<td>APPLICATION</td>
<td>(FUNCTION-SPECIFIER FORMULA*)</td>
</tr>
<tr>
<td>FUNCTION-CONSTANT</td>
<td>#'FUNCTION-SPECIFIER</td>
</tr>
<tr>
<td>FORMULA</td>
<td>CONSTANT</td>
</tr>
</tbody>
</table>

2.9. LAMBDA CALCULUS AND PURE LISP

CONSTANT := NUMBER | 
'S-EXPRESSION

BUILTIN-FUNCTION := SYMBOL

VARIABLE := SYMBOL

ABSTRACTION := (lambda (VARIABLE*) FORMULA)

FUNCTION-SPECIFIER := BUILTIN-FUNCTION | ABSTRACTION

APPLICATION := (FUNCTION-SPECIFIER FORMULA*)

FUNCTION-CONSTANT := #'FUNCTION-SPECIFIER

FORMULA := CONSTANT | FUNCTION-CONSTANT | VARIABLE | APPLICATION
<table>
<thead>
<tr>
<th><strong>CONSTANTS</strong></th>
<th>55</th>
<th>5.5</th>
<th>-123</th>
<th>1/3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'a b c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'(a (b c) (d . e))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BUILTIN-FUNCTIONs</strong></td>
<td>car</td>
<td>+</td>
<td>funcall</td>
<td></td>
</tr>
<tr>
<td><strong>VARIABLEs</strong></td>
<td>x</td>
<td>y</td>
<td>z3</td>
<td></td>
</tr>
<tr>
<td><strong>ABSTRACTIONs</strong></td>
<td>(lambda (x) (* 2 x))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(lambda (x) (car (cdr x)))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>APPLICATIONs</strong></td>
<td>(car '(a b c))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(* 2 x)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(car (cdr x))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>((lambda (x) (* 2 x)) 33)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>((lambda (x) (car (cdr x))) '(a (b c) d))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FUNCTION-CONSTANTS</strong></td>
<td>#'car</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#'(lambda (y) (* 5 y))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2: Grammar of Pure LISP
2.9. *LAMBDA CALCULUS AND PURE LISP*

Lambda calculus does not perform evaluation of expressions in the same way as computer languages. Rather, lambda calculus uses rules for simplifying formulas. These rules are called reduction rules, and the formulas are said to be reduced, rather than evaluated.

Here are examples of the various reductions:

**β-reduction:**
\[(\text{lambda} \ (x) \ (* \ 2 \ x)) \ 33 \xrightarrow{} (* \ 2 \ 33)\]

**α-congruence:**
\[(\text{lambda} \ (x) \ (* \ 2 \ x)) \xrightarrow{} (\text{lambda} \ (y) \ (* \ 2 \ y))\]

**funcall-reduction:**
\[(\text{funcall} \ #'(\text{lambda} \ (x) \ (* \ 2 \ x)) \ 33) \xrightarrow{} ((\text{lambda} \ (x) \ (* \ 2 \ x)) \ 33)\]

**builtin-* operator-reduction:**
\[(* \ 2 \ 33) \xrightarrow{} 66\]

- **β-reduction** is just substitution of arguments for the variables in an abstraction.
- **α-congruence** is just changing the names of the arguments in an abstraction. Such name changes to not change the meaning of the abstraction. **α-congruence** is not actually a reduction rule: it is a means of telling when two formulae are equal (mathematically speaking, it defines an equivalence relation on formulae). When we say that two formula are equal below, what we actually mean is that each can be converted to the other using **α-congruence**.

**Funcall-reduction** arises because COMMONLISP does not allow a variable to be used as a function specifier, but requires that a **funcall** application be used instead. Similarly, COMMONLISP requires that abstractions be made into function constants by the **#’** operator before they can be formulae. (Lambda calculus abstraction corresponds precisely to COMMONLISP closures.) **Funcall-reduction** reduces an expression of the form:
\[(\text{funcall} \ #’(\text{lambda} \ (x) \ (* \ 2 \ x)) \ 33) \xrightarrow{} ((\text{lambda} \ (x) \ (* \ 2 \ x)) \ 33)\]

Other forms of lambda calculus simply allow variables to be function-specifiers and abstractions to be formulae and have neither **funcall** nor **#’**.

Each builtin function has its own reduction rules, such as those for the * function exemplified above.

(Strictly speaking, we were sloppy in stating the β-reduction and α-congruence rules above, but as lambda calculus is not a primary topic of this course, we will not rectify this. Our sloppyness concerns the fact that in formula such as
\[ ((\text{lambda} \ (x) \ ((\text{lambda} \ (x) \ (* \ 2 \ x)) \ (+ \ 1 \ x))) \ 8) \]

there are actually two different and distinct variables named \( x \), and these must be distinguished in applying \( \beta \)-reductions and \( \alpha \)-congruence.)

An important fact about the reduction rules for lambda calculus is that the order in which they are applied makes no difference in the outcome. This is called the Church-Rosser Theorem, and is proved using the nature of the reduction rules. It is analogous to saying that the order of evaluation is not important.

(The last statement is also sloppy. Some formulae can be infinitely reduced: reductions can be chosen so that there is always yet another reduction possible. Some of these formulae can also be reduced by a different choice of reductions to something irreducible, that is, a formula that cannot be further reduced except by \( \alpha \)-congruence. What the Church-Rosser Theorem actually says is that if a formula \( F \) is reducible to \( F_1 \) by one set of reductions and to \( F_2 \) by another set of reductions, then there is a formula \( G \) such that both \( F_1 \) and \( F_2 \) can be reduced to \( G \). Then if, for example, \( F_1 \) is irreducible, it must equal \( G \) (modulo \( \alpha \)-congruence), and if both \( F_1 \) and \( F_2 \) are irreducible, they must be equal (modulo \( \alpha \)-congruence).)

In order to preserve the Church-Rosser Theorem in Pure LISP, special forms, macros, and destructive functions are not allowed in Pure LISP. One can redefine \textit{if} to be compatible with Pure LISP by premitting all its arguments be evaluated (i.e. reduced), with the second or third argument value being selected according to the value of the third argument. Since in Pure LISP no evaluatable argument can have any side-effect, such as setting a variable or terminating the program, the redefined \textit{if} has effectively the same semantics as the normal LISP \textit{if} that prohibits evaluation of one of its arguments.

(Note that there is no such thing as an “error” in Pure LISP. An expression such as \( (+ \ 'X \ 'Y) \) is merely irreducible: there is no reduction rule to simplify it.)

Operations such as \textit{setf} and destructive LISP functions, that make order of evaluation important, are called \textit{imperative}. Operations that leave order of evaluation unimportant are called \textit{functional}. Imperative functions are definitely excluded from Pure LISP. But there is no real problem with adding things like macros, \textit{flet}, \textit{cond}, etc. to Pure LISP, though one would have to extend the syntax given above.

Note that recursive functions are not part of lambda calculus or Pure LISP. If we add \textit{defun} and \textit{labels} (which is the recursive form of \textit{flet}) to Pure LISP, recursive functions may be defined, but we still have a system in which evaluation may be performed by reduction in an order which is arbitrary (except for issues involving termination). Languages that are essentially lambda calculus (with constants and non-imperative builtin functions) extended with recursive function definition are called \textit{functional programming languages}. The \textit{do} macro can expand into a recursive function (see \textit{Recursion and Iteration in LISP}) and so can be included in a functional programming language. Imperative functions cannot be included in a functional programming
2.10. **LISP PROJECTS**

language (but real-world functional programming languages, such as SML, include them anyway, and are therefore impure).

Functional programming languages are being researched actively today because programs that are less sensitive to execution order can be run faster by parallel computers.

**2.10 LISP Projects**

Specifications are given below for various functions. You may be asked to write either the detailed design or the code for any of these functions. We could also ask for preliminary design, but because these functions are so small, we will assume that the preliminary and detailed design can be done together with the results written in the format of a detailed design.

Detailed design should be written in a file named `functions.dd` which begins with a header of the form:

```
;;; {one-line description of file}
;;;
;;; File:         functions.dd
;;; Assignment:  {1, 2, etc.}
;;; Author:       {Your name} ({Your email address})
;;; Collaborators: {Collaborators names}
;;; Version: 1
```

Here you fill in the bracketed parts. If you happen to submit a second version of the file, increment the version number. You are permitted to collaborate on the detailed design files in this project (you should not assume that collaboration is allowed in other projects), but must name your collaborators on the indicated file header line. Collaboration does not mean that two students may submit the same file: you should type in your own file which should show some individualism.

Code should be submitted in a file named `functions.lsp` with a header of the same format but different filename. Collaboration is not permitted for code files: the Collaborators field should be empty.

The detailed design should be written before the code, and the code file should start out as a copy of the detailed design file with a modified header. As code is written, design mistakes may be discovered, and the design file may become out-of-date. In this course you need not go back to the design file and fix it: you may leave it out-of-date.

You may be asked to write tests for the code you have written. Tests of the code should be submitted in a file named `functions.in` with a header of the same format but different file name. Collaboration is permitted for test files just as for design files.

Detailed rules concerning collaboration are given elsewhere. Be sure you know them if you do any collaboration.
The results of running the tests should be obtainable by running the UNIX make command using a Makefile which you have been given. Those results will be in the file functions.out which records the output which would appear if you typed functions.in into LISP after loading functions.lsp.

The following is an example. Suppose we are asked to write a function named dotted that lists all the non-nil atoms in an s-expression which are cdr’s of some dotted pair in the s-expression. Suppose further than the atoms are to be listed in the order they occurred in the s-expression, and in the interests of efficiency, only one cons-cell is to be created for each atom in the output list.

The detailed design that would be placed in functions.dd might be:

```
; Function to return a list of all the non-nil atoms that
; are cdr’s anywhere within the s-expression expr, in the
; order these atoms appear in the s-expression.
;
(defun dotted (expr)
  (nreverse (internal-dotted expr nil)))
```

; Function that does what dotted does, but prepends the
; atoms found to the result list and returns that list,
; so the atoms are in reverse order on the list.
;
(defun internal-dotted (expr result)
  (cond
    ((expr is an atom)
     {return result})
    ((cdr of expr is nil)
     {replace expr by (car expr) and call recursively})
    ((cdr of expr is a non-nil atom)
     {replace result by (internal-dotted (car expr) result);
      prepend (cdr expr) to new result and return result; })
    (else
     {replace result by (internal-dotted (car expr) result);
      replace expr by (cdr expr);
      call recursively;})
```

You can invent your own pseudo programming language within the { } brackets, as long as it is clear.

You are required to explain each function in comments (e.g. ; Function that ...) so a
person trying to use the function need not read the function code to find out what the function does. You may treat the introductory part of the \texttt{defun} statement, through the argument list, as part of these comments, so if you use descriptive argument names you may save yourself some work. You are similarly required to explain each new variable you introduce, if its use is not obvious from its name.

However, you should avoid writing comments that repeat information obvious to an experienced programmer reading the code.

The code placed in \texttt{functions.lsp} might be:

\begin{verbatim}
; Function to return a list of all the non-nil atoms that
; are \texttt{cdr}'s anywhere within the s-expression \texttt{expr}, in the
; order these atoms appear in the s-expression.
;
(defun dotted (expr)
    (nreverse (internal-dotted expr nil)))

; Function that does what dotted does, but prepends the
; atoms found to the result list and returns that list,
; so the atoms are in reverse order on the list.
;
(defun internal-dotted (expr result)
    (cond
        ((atom expr) result)
        ((null (cdr expr))
            (internal-dotted (car expr) result))
        ((atom (cdr expr))
            (cons (cdr expr) (internal-dotted (car expr) result)))
        (t
            (internal-dotted (cdr expr)
                (internal-dotted (car expr) result))))
\end{verbatim}

This code is created by making a copy of the detailed design file and replacing the bracketed statements.

One test in \texttt{functions.in} might be:

\begin{verbatim}
(dotted '(a (b . c) (k j . m) (b (d . g))))
\end{verbatim}

and when run this might yield the following in \texttt{functions.out}:

\begin{verbatim}
-> (dotted '(a (b . c) (k j . m) (b (d . g))))
(C M G)
\end{verbatim}
The example we have just given is one of the harder problems in this project. For such problems, doing a detailed design carefully usually pays off.

2.10.1 Project: Defining Functions in LISP

For each of the following functions, you may assume that the arguments passed are of the correct form. That is, you do not have to worry about error-checking input. Feel free to use either iteration or recursion for the exercises, unless a certain method is specified for a problem. You must comment any non-obvious code in your routines, and explain each function in comments just before the function definition.

1. Write a function new-list which takes one argument (a non-negative integer) and returns a list of that length, containing all t’s. For example, (new-list 5) should evaluate to (T T T T T).

2. Write a function 2nd-to-last which takes one argument (a list) and returns the second to last s-expression in the list. For example,

   (2nd-to-last '(a b (c)))

   should evaluate to B. You may assume the list argument has at least two elements.

3. Write a two argument function my-member which is just like member with :test #'=, i.e. equality of numbers is tested. You may not use the member function.

4. Write a function remove-2nd-to-last which takes one argument (a list) and returns the list without its second to last s-expression. For example,

   (remove-2nd-to-last '(a b (c)))

   should evaluate to (A (C)). You may not use the built-in functions delete or remove, and you may assume the list argument has at least two elements. Your function must not destroy its input list (you should make a copy).

5. Write a recursive function length-r which takes one argument (a list) and returns its length. For example, (length-r '(a b (c))) should evaluate to 3. You may not use the built-in function length.

6. Write an iterative function length-i, following the instructions of the previous problem. Feel free to use the iterative control structure with which you are most comfortable.
7. Write a recursive function \texttt{print-r} which takes one argument (a list) and prints out each element of the list separated by a space. When done, it should send a carriage return and return \texttt{t}. (Review the use of \texttt{print}, \texttt{princ} and \texttt{terpri} before trying to do this problem.) For example:

\begin{verbatim}
\rightarrow (print-r '(a (c)))
A (C)
\texttt{T}
\end{verbatim}

Your function must \textbf{not} put a space just in front of the carriage return.

8. Write an iterative function \texttt{print-i} which performs exactly like \texttt{print-r}.

9. Write a function \texttt{count-sublists} which takes one argument (a list) and counts the number of sublists in the list. For example, the list \((a b c)\) contains no sublists; the list \(((a (b c)) (d e))\) contains three sublists — two at the top-level and one at the next level. Here \texttt{nil} is a sublist: \((a \texttt{nil} b)\) contains one sublist.

10. Write a function \texttt{assoc-compatible} which takes two association lists as its arguments and determines whether they are “compatible.” Two association lists are compatible if they do not associate the same atom with different values. You should use \texttt{equal} to test two values for equality. For example:

\begin{verbatim}
\rightarrow (assoc-compatible '((x 7) (y 83) (z 29)) '((a 5) (b 62)))
\texttt{T}
\rightarrow (assoc-compatible '((x 7) (y 83) (z 29)) '((w 5) (z 29)))
\texttt{T}
\rightarrow (assoc-compatible '((x 7) (y 83) (z 29)) '((x 7) (z 103)))
\texttt{NIL}
\end{verbatim}

11. Write a recursive function \texttt{remove-doubles} which takes one argument (a list) and returns a list that contains each element only once. You should not use the built-in functions \texttt{delete}, \texttt{remove}, or \texttt{remove-duplicates}. For example:

\begin{verbatim}
\rightarrow (remove-doubles '(a b (c d) (c d) b d (c)) c (c))
(C (C) D (C D) B A)
\end{verbatim}

Your function must \textbf{not} destroy its input list (you should make a copy). Your function may change the order of elements in the list, and you may find it easiest to reverse the order. You should use \texttt{equal} to test two elements for equality.

12. Write a function \texttt{add-to-end} which takes two arguments (an atom and a list) and adds the atom to the end of every sublist contained in the list (at any level!). For example:
13. Write a function `typer` which takes one argument (an s-expression) and returns an s-expression having the same “shape” but with only the symbols `number`, `symbol`, and `nil` — corresponding to the types of the original atoms in the list. `Nil` is a special case with type `nil`. A number is anything satisfying `numberp`. For example:

```lisp
-> (typer '(fred ate 5 socks . Wow!))
(SYMBOL SYMBOL NUMBER SYMBOL . SYMBOL)
-> (typer '((catch 22.0) nil ((3 dog night)) 6))
((SYMBOL NUMBER) NIL ((NUMBER SYMBOL SYMBOL)) NUMBER)
```

14. Write a function `weird-eval` which takes one argument that is an s-expression which contains only numbers and the operators `+` and `*`, and is a well formed algebraic expression, but may have any number of operands, including zero, for an operator. However, `weird-eval` reverses the meaning of the two operators. For example:

```lisp
-> (weird-eval '(* 5 9))
14
-> (weird-eval '(+ 5 9))
45
-> (weird-eval '(*))
0
-> (weird-eval '+))
1
```

15. Write a function `my-reverse` that does exactly what `reverse` does. You may not use `reverse`, of course. In the interest of efficiency, you should create no more than one new cons-cell for each element of the input list. You must not destroy the input list: `reverse` is non-destructive.

For each of the following functions, you are expected to handle bad input by calling the `error` function. Note that this function requires a character string argument in double quotes, as in:

```lisp
(error "argument is not a list")
(error "argument ~S is not a number" arg)
```
2.10. LISP PROJECTS

Note that error is a bit like printf in C and can use ~S to insert an s-expression in the output just like printf can use %s to insert a string. However, you are not required to use this feature.

When you write function in lines that intentionally invoke error, you may have to follow special instructions to recover from the error. For example, in Ibuki or Kyoto COMMONLISP you may have to write :q on the line after an s-expression which calls error when it evaluates. You will be given separate instruction on how to proceed with this matter.

You should not, however, concern yourself with checking whether cdr erroneously returns a non-nil atom. You may assume cdr returns a list when only that possibility is error-free.

Include an explanation of each function in comments, and comment any non-obvious code in your routines, including error checking code.

16. Write a function ratio that takes one argument (a list of numbers) and returns the ratio of the biggest number in the list to the smallest number in the list.

17. Write a function shuffle which takes two lists as its arguments and returns a new list which is the result of alternating the elements of the two lists. If the lists are of unequal length, the remainder of the longer length should be at the end of the result. For example:

\[
\begin{align*}
\text{-> (shuffle '(a b c) '(x y z))} & \quad (A \ X \ B \ Y \ C \ Z) \\
\text{-> (shuffle '(a b c d) '(x y))} & \quad (A \ X \ B \ Y \ C \ D) \\
\text{-> (shuffle '(a (b c)) '(v w (x y) z))} & \quad (A \ V \ (B \ C) \ W \ (X \ Y) \ Z)
\end{align*}
\]

18. Write a function fringe which takes a list as its only argument and returns a list whose elements are all the atoms appearing in the original list, in the same left-to-right order, but with all of them now appearing at the top level. (That is, given a tree structure, fringe returns the list of the leaves of the tree, eliminating all inner parentheses.) For example:

\[
\begin{align*}
\text{-> (fringe '((1 (2) (3 4 2)) (5 3) 6))} & \quad (1 \ 2 \ 3 \ 4 \ 2 \ 5 \ 3 \ 6)
\end{align*}
\]

19. Write a function my-assoc which takes two arguments (an atom and a list of sublists) and returns the first sublist of the list in which the atom is the first element. This works just like the built-in function assoc, which you may not use in writing this routine. If no sublist has the atom as its first element, then my-assoc should return nil. If the second argument is not a list of sublists, my-assoc should call error. For example:
20. Write a function `powerset` which takes as its only argument a set of atoms and returns a list of all the possible distinct sets which can be formed from those atoms. (A set is a list which contains no duplicate expressions — e.g., `(a b 2)` would be a set of atoms, while `(a b c b)` would not.) If the argument is not a set of atoms, `powerset` should call `error`. For example:

```lisp
-> (powerset '(a b 2))
(nil (A) (B) (a b) (2) (A 2) (B 2) (A B 2))
-> (powerset '(A B C B))
Error: argument is not a set of atoms
-> (powerset '(a (b 2)))
Error: argument is not a set of atoms
```

The order of the sets in the return value is not important, but no duplicate sets (like `(a b)` and `(b a)`) should be returned. The `eql` function should be used to test equality of elements.

21. Write a function `my-nreverse` that does exactly what `nreverse` does. You may not use `nreverse`, of course. You should not create any new cons-cells, but you should be destructive instead.

22. Write a function `make-number-sequence` that takes a number argument `n` and makes an ordered list of all non-negative integers not greater than `n`. For example:

```lisp
-> (make-number-sequence 5.1)
(0 1 2 3 4 5)
-> (make-number-sequence -0.1)
NIL
```

In the interest of efficiency, `make-number-sequence` should create at most one cons-cell for each element of the resulting list. You should consider making the list in the reverse order and applying `nreverse`.\[\]
Chapter 3
Properties and Pattern-Matching
by Jason Abrevaya and Harry Lewis

3.1 Properties in LISP

A feature which separates LISP from many programming languages is the ability it offers to associate properties with objects. While a symbol in LISP can have only a single top-level value associated with it, it can also have any number of properties associated with it. A person, represented by some symbol, could be associated with such properties as age, sex, hair color, and friends, to name but a few. These characteristics are known as property names, and each person can have property values associated with these property names. For instance, we can say that Fred is 40 years old, is male, has black hair, and has (Barney Betty Wilma) as his friends.

In LISP, there are two basic functions (putprop and get) which allow us to implement such a notion of properties. The function putprop takes three arguments — a symbol, a property value, and a property name — and changes the property (as specified by the property name) of the symbol to the specified property value. In the above example, to indicate that Fred is 40 years of age, we would do:

```
-> (putprop 'Fred '40 'age)
40
```

Note: If you are using a COMMONLISP system in which putprop is not defined, it can be defined as follows:

```
-> (defun putprop (sym pval pname)
   (setf (get sym pname) pval))
```

We can also specify the other properties for Fred at this point:
-> (putprop 'Fred 'male 'sex)
  male

-> (putprop 'Fred 'black 'hair-color)
  black

-> (putprop 'Fred '(Barney Betty Wilma) 'friends)
  (Barney Betty Wilma)

To access the values associated with properties, we use the function get, which takes two arguments — a symbol and a property name — and returns the appropriate property value. For example:

-> (get 'Fred 'hair-color)
  black

-> (get 'Fred 'sex)
  male

If no value has been associated with a particular property of a symbol, get will return nil:

-> (get 'Fred 'height)
  nil

There is, however, no way to distinguish between a property which has not yet been defined and one whose value is nil. For instance, if the friends property for Fred is nil, either (1) the friends property has yet to be putprop’d, or (2) Fred has no friends.

3.2  Implementing Knowledge Networks with Properties

One very natural application of properties is the development of knowledge networks. The type of knowledge network which we’ll be outlining is a simple one known as an ako-network (where “ako” stands for “a kind of”). The goal is to represent a hierarchical tree of nodes, in which each node represents some object and each of the branches signify that nodes are related by an “ako” relationship. As an example, consider the following diagram:
In this tree, we see, for instance, that a dog is a kind of mammal, which is a kind of animal, which is a kind of living-thing. A simple way of setting up such a structure in LISP is to use ako as a property name and define the ako property appropriately for each of the nodes in the tree. For instance, to indicate that dog is a kind of mammal, the appropriate expression would be (putprop 'dog 'mammal 'ako). Each of the nodes in the tree will have its ako property defined to be its immediate predecessor in the tree structure. Since living-thing has no predecessor (as the topmost node in the tree, it is usually known as the root), we'll just define its ako property to be nil. Here is the complete LISP code needed to set up the desired knowledge network:

```lisp
(putprop 'living-thing nil 'ako)
(putprop 'animal 'living-thing 'ako)
(putprop 'plant 'living-thing 'ako)
(putprop 'bird 'animal 'ako)
(putprop 'mammal 'animal 'ako)
(putprop 'reptile 'animal 'ako)
(putprop 'grass 'plant 'ako)
(putprop 'tree 'plant 'ako)
(putprop 'dog 'mammal 'ako)
(putprop 'human 'mammal 'ako)
(putprop 'lizard 'reptile 'ako)
(putprop 'crocodile 'reptile 'ako)
(putprop 'jack 'human 'ako)
(putprop 'jill 'human 'ako)
(putprop 'fido 'dog 'ako)
```
Having this network, we might want to ask questions like: “Is a lizard a plant?” or “Is a tree a living-thing?” The answers to such questions are not always immediately available since our property definitions only deal with first-order ako relations. That is, each object is defined to be a kind of another object, which is just one level above it in the hierarchical structure. We can, however, write a function ako? which will answer such questions in all cases:

```
(defun ako? (obj1 obj2)
  (cond ((equal obj1 obj2) t)
        ((null obj1) nil)
        (t (ako? (get obj1 'ako) obj2))))
```

```
-> (ako? 'lizard 'plant)
nil

-> (ako? 'tree 'living-thing)
t
```

Another item of interest might be to determine the most immediate ancestor of any two objects in the network. For instance, the most immediate ancestor of lizard and dog is animal. We can utilize ako? to write a new function find-ancestor:

```
(defun find-ancestor (obj1 obj2)
  (cond ((ako? obj1 obj2) obj2)
        ((ako? obj2 obj1) obj1)
        (t (find-ancestor (get obj1 'ako) obj2))))
```

```
-> (find-ancestor 'lizard 'dog)
animal

-> (find-ancestor 'living-thing 'jack)
living-thing

-> (find-ancestor 'animal 'chop-suey)
nil
```
One of the major reasons for creating an ako-network is that it provides a convenient way to associate a property with an entire class of objects without having to associate the property with each object in the class. For instance, we might want to establish the fact that all mammals are covered with hair. Ideally, we should be able to attach such a property to the mammal object and have this property automatically associated with all types of mammals (like dog and jack) via a notion of "inheritance." To do this, we define a function ako-get which works like get except that it can also retrieve “inherited” properties:

```
(defun ako-get (sym pname)
  (cond ((null sym) nil)
        ((get sym pname))
        (t (ako-get (get sym 'ako) pname))))
```

- (putprop 'mammal 'hair 'bodycover)
  hair
- (putprop 'human '2 'legs)
  2
- (ako-get 'jack 'bodycover)
  hair
- (ako-get 'jack 'legs)
  2
- (ako-get 'tree 'legs)
  nil

In talking about knowledge networks, we’ve actually sidestepped a potential stumbling block by not considering the special case of circular relations. In the ako-network developed above, we could create such a circular relation with the expression (putprop 'living-thing 'fido 'ako). With this new relationship defined, both (ako? 'fido 'living-thing) and (ako? 'living-thing 'fido) would evaluate to t. The existence of such a circular relation in an ako-network creates the possibility for infinite looping within each of the three functions we’ve written. For instance,
would all result in infinite loops. In order to allow for circular relations, we need to keep track of the objects which have been checked by our function to prevent infinite looping. Here is the appropriate fix for `ako?

; version of ako? that avoids infinite looping in the case of circular relations
(defun new-ako? (obj1 obj2)
  (do ((objs-seen nil (cons temp objs-seen))
       (temp obj1 (get temp 'ako)))
      ((or (null temp) (member temp objs-seen)) nil)
      (if (equal temp obj2) (return t))))

The fixes for `ako-get` and `find-ancestor` are similar, and left to the reader.

### 3.3 Pattern-Matching

Pattern matching is a powerful tool for many computer applications. Many operating systems, for instance, use a form of pattern-matching to process file-oriented commands. A user might be able to get a listing of all files having the suffix “.txt” or all files that have four-letter names. Pattern-matching is also useful for building up an understanding of language since it enables the computer to search for certain key words within a user’s input in order to decipher what the user is trying to say.

In what follows, we will be implementing some simple pattern-matching within LISP. Each of our matching functions will take two arguments — a pattern and an item of data — and return \( t \) if the data matches the pattern, \( nil \) otherwise. This form of pattern-matching may seem far more simplistic than the applications described above, but it can actually be used as the basis for developing such applications. At the end of this section, we’ll use our patternmatcher to develop some basic language-understanding skills for the computer.

As a convention, both of the arguments to our matching functions will be lists. Our first pattern-matching function is a simple one:

(defun match1 (ptrn dat)
  (cond ((and (null ptrn) (null dat))
         \( t \))
        ((equal (car ptrn) (car dat))
         (match1 (cdr ptrn) (cdr dat)))))
It turns out that \texttt{match1} is not all that interesting. It works in precisely the same way as LISP’s built-in function \texttt{equal}:

\begin{verbatim}
-> (match1 '(a b c) '(a b c))
t
-> (match1 '(a b c) '(a (b) c))
nil
\end{verbatim}

To be a little more adventurous, we allow patterns to contain a special character \texttt{?}. Each occurrence of \texttt{?} in the pattern is considered to be a “match” with the corresponding expression in the item of data, no matter what that expression may be. For instance, the pattern (a ? c) would match with (a b c), (a (b d) c), (a nil c), etc. Here is the code for our new pattern-matcher:

\begin{verbatim}
(defun match2 (ptrn dat)
  (cond ((and (null ptrn) (null dat)) t)
        ((or (equal (car ptrn) '?)
             (equal (car ptrn) (car dat)))
             (match2 (cdr ptrn) (cdr dat))))
)
\end{verbatim}

The observant reader will notice a slight error in this code. Consider what happens when \texttt{?} is the last expression in a pattern. In this case, \texttt{match2} will match the pattern with an item of data even if that item does not have an expression to correspond with the \texttt{?} (i.e., (match '(a b ?) '(a b)) evaluates to \texttt{t}). In order to fix this bug, we add another base case to \texttt{match2}:

\begin{verbatim}
(defun match3 (ptrn dat)
  (cond ((and (null ptrn) (null dat)) t)
        ((or (null ptrn) (null dat)) nil) ; add’l base case
        ((or (equal (car ptrn) '?)
             (equal (car ptrn) (car dat)))
             (match3 (cdr ptrn) (cdr dat))))
)
\end{verbatim}

We now allow for another special symbol \texttt{*} to appear in patterns. Unlike \texttt{?}, which matches with only a single expression, \texttt{*} matches with any number (including zero) of expressions. For instance, (a * c) matches with (a c), (a b c), (a b d c c), etc. This addition requires a bit more code since we don’t know, \textit{a priori}, how many expressions to try to match a \texttt{*} against. As a result, we must try all possibilities:
(defun match4 (ptrn dat)
  (cond ((and (null ptrn) (null dat)) t)
        ((equal (car ptrn) ’*)
         (cond ((and (null dat) ; failure if no data
                     (not (null (cdr ptrn))) ; left and more ptrn
                     nil) ; follows the *
                 ((match4 (cdr ptrn) dat)) ; match * vs. 0 exprs
                 ((match4 ptrn (cdr dat))) ; match * vs. 1+ exprs
                 ((or (null ptrn) (null dat)) nil)
                 ((or (equal (car ptrn) ’?)
                   (equal (car ptrn) (car dat)))
                   (match4 (cdr ptrn) (cdr dat)))))
  (match5 (cdr ptrn) (cdr dat))))

Notice that the second base case has been moved further down in the code so that it appears after the section dealing with the * character. This change handles patterns having * as their last expression (e.g., (a b *)).

We now have our basic pattern-matcher. To make it a little more useful, we would like to record the expressions that are matched to a given ? or * symbol. To allow this, we offer the option of replacing a ? or * symbol by a two-element list, having a ? or * as its first element and some symbol as its second element. The portion of data that matches the * or ? symbol will be stored in the value binding of the symbol specified in the two-element list. For instance, if the pattern (the ball is (? color)) is matched against the item of data (the ball is red), our pattern-matcher should return a value of t and, as a side-effect, bind the value red to the symbol color. We first implement this modification for only the ? symbol:

(defun match5 (ptrn dat)
  (cond ((and (null ptrn) (null dat)) t)
        ((equal (car ptrn) ’*)
         (cond ((and (null dat)
                      (not (null (cdr ptrn)))
                     nil)
                 ((match5 (cdr ptrn) dat))
                 ((match5 ptrn (cdr dat)))
                 ((or (null ptrn) (null dat)) nil)
                 ((or (equal (car ptrn) ’?)
                   (equal (car ptrn) (car dat)))
                   (match5 (cdr ptrn) (cdr dat))))
  (match5 (cdr ptrn) (cdr dat))))

We now have our basic pattern-matcher. To make it a little more useful, we would like to record the expressions that are matched to a given ? or * symbol. To allow this, we offer the option of replacing a ? or * symbol by a two-element list, having a ? or * as its first element and some symbol as its second element. The portion of data that matches the * or ? symbol will be stored in the value binding of the symbol specified in the two-element list. For instance, if the pattern (the ball is (? color)) is matched against the item of data (the ball is red), our pattern-matcher should return a value of t and, as a side-effect, bind the value red to the symbol color. We first implement this modification for only the ? symbol:
((and (listp (car ptrn))
  (equal (car (car ptrn)) '?))
 (set (car (cdr (car ptrn))) (car dat))
 (match5 (cdr ptrn) (cdr dat))))}

We now implement the comparable modification for the * symbol. Since * can match with more than one expression, we will bind the appropriate symbol to the list of expressions that match with *. For example, the pattern (the ball is (* what)) matched against the data (the ball is red and round) would bind the value (red and round) to the symbol what:

(defun match6 (ptrn dat)
  (cond ((and (null ptrn) (null dat)) t)
        ((equal (car ptrn) '*)
         (cond ((and (null dat)
                   (not (null (cdr ptrn))))
                nil)
               ((match6 (cdr ptrn) dat))
               ((match6 ptrn (cdr dat)))))))

((and (listp (car ptrn))
 (equal (car (car ptrn)) '*))
 (cond ((and (null dat)
           (not (null (cdr ptrn))))
        nil)
       ((match6 (cdr ptrn) dat))
       ((match6 ptrn (cdr dat)))))))

((or (null ptrn) (null dat)) nil)

((or (equal (car ptrn) '?)
     (equal (car ptrn) (car dat)))
 (match6 (cdr ptrn) (cdr dat))))
The final addition we’ll make is a restriction mechanism, to enable further filtering of data through the use of LISP predicates (be they built-in or user-defined). (Note: To qualify as a predicate, a function must take only one argument and return a non-nil value (usually t) if and only if the argument satisfies the predicate.) With this addition, the pattern (the number (filter (? num) numberp evenp) is even) would match against (the number 10 is even) but not against (the number a is even) or (the number 9 is even). The restriction mechanism is invoked whenever the pattern-matcher sees the symbol filter as the first element of a list. The next element of this list should be either the ? symbol or a two-element list that begins with a ?, depending on whether the matched data should be saved in a global binding. Finally, any number of predicates should follow; the pattern-matcher will accept only that data which satisfies all of the predicates. Here is the final version of our pattern-matcher:

```lisp
(defun match (ptrn dat)
  (cond ((and (null ptrn) (null dat)) t)
        ((equal (car ptrn) '*)
         (cond ((and (null dat)
                     (not (null (cdr ptrn))))
                nil)
               ((match (cdr ptrn) dat))
               ((match ptrn (cdr dat)))))
        ((and (listp (car ptrn))
               (equal (car (car ptrn)) '*))
         (cond ((and (null dat)
                     (not (null (cdr ptrn))))
                nil)
               ((match (cdr ptrn) dat)
                (set (car (cdr (car ptrn))) nil)
                t)
               ((match ptrn (cdr dat))
                (set (car (cdr (car ptrn)))
                     (cons (car dat)
                           (eval (car (cdr (car ptrn))))))))
        t)))
```
3.4. AN EXAMPLE USING PATTERN-MATCHING

To give a short illustration of the usefulness of the pattern-matcher that we have developed, we offer a simplified version of a program called ELIZA that was originally developed by Joseph Weizenbaum. ELIZA is a “computerized psychologist” that conducts a question-and-answer session with
the user. Our program, `doctor`, conducts such a session, albeit quite shallow, by using `match`:

```lisp
(defun doctor ()
  (princ "Speak up!"
  (do ((user-input) (mother))
      (nil)
      (terpri) (terpri)
      (setq user-input (read))
      (cond ((match '(* I am worried (* blah-blah)) user-input)
           (terpri)
           (princ (append '(How long have you been worried)
                          blah-blah)))
           ((match '(I wish (* blah-blah)) user-input)
            (terpri)
            (princ (append '(Tell me why you wish)
                          (first-person-to-second blah-blah)))
           ((match '(* mother *) user-input)
            (setq mother t)
            (terpri)
            (princ "Tell me more about your family..."))
           ((match '(* computers *) user-input)
            (terpri)
            (princ "Do machines frighten you???")
           ((or (match '(no) user-input)
                (match '(yes) user-input))
            (terpri)
            (princ "Please do not be so short with me!")
           (match '(* (filter ? badwordp) *) user-input)
            (terpri)
            (princ "Please do not use words like that!!!")
           (mother
            (setq mother nil)
            (terpri)
            (princ "Earlier you spoke of your mother..."))))
```

; doctor uses pattern-matching to conduct a psychological conversation with the user
3.4. AN EXAMPLE USING PATTERN-MATCHING

(t
  (terpri)
  (princ "I am sorry -- our time is up!")
  (terpri)
  (return 'Good-bye))))))

; badwordp is a predicate which returns t iff word is a bad word
(defun badwordp (word)
  (member word '(gosh sexy darn heck legs turkey)))

; first-person-to-second converts a phrase spoken in the
; first person to one spoken in the second person
(defun first-person-to-second (phrase)
  (cond ((null phrase) nil)
        ((equal (car phrase) 'I)
          (cons 'you (first-person-to-second (cdr phrase))))
        ((equal (car phrase) 'my)
          (cons 'your (first-person-to-second (cdr phrase))))
        ((equal (car phrase) 'you)
          (cons 'me (first-person-to-second (cdr phrase))))
        (t
          (cons (car phrase)
                (first-person-to-second (cdr phrase)))))))

Here is a sample session with doctor:
-> (doctor)
Speak up!

(I wish I could be somewhere else)
(Tell me why you wish you could be somewhere else)

(because I am worried about passing this course)
(How long have you been worried about passing this course)

(ever since my mother first warned me about the professor)
Tell me more about your family...

(they are all into computers and leather -- I think)
Do machines frighten you???

(I don’t know)
Earlier you spoke of your mother...

(yes)
Please do not be so short with me!

(sorry -- you are one heck of a smart program)
Please do not use words like that!!!

(words like what???)
I am sorry -- our time is up!
Good-bye

Our doctor program certainly wouldn’t be confused with an actual psychologist, but it does demonstrate that we can, in some sense, make the computer “understand” the patient’s sentences through the use of pattern-matching. The more extensive the vocabulary that is supplied by the programmer, the better the “understanding” of the computer will be. The reader may find it interesting to add to the code already supplied in order to create a more educated doctor.

3.5 Pattern Matching Projects

3.5.1 Project: Sentence Generator

3.5.1.1 A Simple Grammar

A grammar is really just a formalism for representing constraints about a language. Grammars can, in fact, be considered a “theory” of language since a grammar, by its very nature, contains a description of the constraints that determine whether a particular word sequence is a valid sentence, or whether it is disordered garbage. For our purposes, a grammar will consist of a sequence of rules (to be represented in LISP as list structures).

The following rules define a ridiculous subset of English:
3.5. PATTERN MATCHING PROJECTS

{sentence} ::= {noun-part} {verb-part} . | WHO {verb-part} ?

{noun-part} ::= {article} {modified-noun} | {proper-noun}

{article} ::= A | THE

{modified-noun} ::= {noun} | {adjective} {modified-noun}

{noun} ::= BALL | BOOK | BABY

{adjective} ::= BIG | BLUE | BORING

{proper-noun} ::= JILL | BERTRAND RUSSELL | LARRY BIRD

{verb-part} ::= {transitive-verb} {noun-part}

{transitive-verb} ::= KICKED | KISSED | EXPOUNDED ON

In the above grammar representation, ::= means “is one of the following” and | means “or.”

Words in capital letters (and the punctuation symbols . and ?) are the actual words that will
appear in a sentence generated by the grammar. These are frequently referred to as the terminal symbols of the language. Words in lower-case letters which are enclosed in squiggly braces (e.g., {noun}, {verb-part}) are used to represent syntactic or grammatic categories (i.e., “parts of speech”) in the language being defined. These are usually called the non-terminal symbols of the language since these symbols will not appear in a sentence generated by the grammar.

In order to generate grammatical sentences that obey the specified rules, we must start with a top (or root) non-terminal symbol — in this case, {sentence}. We then choose one of the alternatives which appears on the right-hand side of the rule whose left-hand side is {sentence}. Notice that {sentence} has two possible right-hand-side alternatives (the sequence {noun-part} {verb-part} , or the sequence WHO {verb-part} ?).

After choosing one of these alternatives, we then scan left-to-right along the symbols of the chosen alternative. If the symbol being considered is a terminal, it can be included immediately in the “sentence.” If, on the other hand, the symbol is a non-terminal, then the rule associated with this non-terminal is chosen, and the above procedure is repeated (recursively!) until only terminal symbols remain. This process is more easily understood by considering a complete example in which we generate a sentence from the illustrated grammar:
In this example, the sentence

JILL KICKED THE BIG BALL.

is generated.

Of course, not every sentence which can be created from the grammar will make perfect sense. But meaning is in the realm of semantics and pragmatics; at this point, we are interested only in syntactic or grammatical correctness. Other possible grammatically correct sentences which could be generated include:

A BLUE BABY KISSED THE BORING BALL.
BERTRAND RUSSELL EXPOUNDED ON THE BIG BORING LARRY BIRD.
JILL KICKED LARRY BIRD.
WHO KICKED THE BIG BLUE BOOK?

One of the most important features to notice about our simple grammar is that it is recursive. The definition of \{modified-noun\} includes as an alternative the sequence \{adjective\} \{modified-noun\}. This means that our grammar is right-recursive since a non-terminal (\{modified-noun\} in this case) appears at the right end of one of its own alternatives. What this rule actually states, then, is that we can have as many \{adjective\}s as we would like preceding a \{noun\} in a \{sentence\}.
3.5. PATTERN MATCHING PROJECTS

3.5.1.2 Representing Grammars in LISP

To represent our grammar rules in LISP, we make use of property lists in order to attach properties to each non-terminal symbol in our grammar. The properties of interest in this example are the right-hand-side alternatives which are indicated in the rules for the non-terminal symbols. For each non-terminal, the rhs (to stand for “right-hand side”) property will be represented by a list of alternatives, where each alternative is itself a list of symbols. For instance, the list representation of the rhs property for sentence would be:

```
( (noun-part verb-part .) (WHO verb-part ?) )
```

To actually associate sentence with this property, the following command would be used:

```
(putprop 'sentence
 ''( (noun-part verb-part | . |) (WHO verb-part ?) ) 'rhs)
```

Two things should be pointed out about this expression:

1. Our LISP representation no longer distinguishes between terminals and non-terminals by putting the latter inside squiggly braces. Rather, we have made the convention that terminals are simply those symbols which have no rhs on their property lists (i.e., a nil-valued rhs property). Since the get function in LISP returns nil if no putprop has been issued, it is not necessary to issue a putprop for each of the terminal symbols.

2. The period must be surrounded by straight slashes to be handled properly by LISP. This change doesn’t require any additional special code. (The question mark should be fine all by itself; if not, treat it as you treated the period.)

Before implementing your sentence generator, you should first represent the grammar given at the beginning of this project so that you can test the generator.

3.5.1.3 Implementing a Sentence Generator

The goal of the sentence generator is to generate pseudo-random sentences from a given grammar. To do this, you should write a function generate which takes a symbol as its argument. If the symbol is a terminal symbol, then generate should just print out the atom, followed by a space. If the symbol is non-terminal, however, generate should randomly select an alternative from the right-hand side of the appropriate grammar rule and recursively apply generate to each component of the alternative.

In writing generate, two built-in LISP functions (nth and random) will be helpful. nth accepts a list argument lst and an integer value n and returns the n-th element in lst. (Be careful, though, because nth considers the first element in a list to be the 0-th element; so, for instance,
(nth 2 'a b c) evaluates to c.) The function random takes an integer argument \( n \) and returns a pseudo-random integer value from 0 to \( n - 1 \), inclusive. (Note: If nth or random are not defined in the version of LISP you are using, please write your own versions of these functions.)

If coded correctly, generate should return a pseudo-random \{modified-noun\} if called with modified-noun as an argument, a pseudo-random \{sentence\} if called with sentence, and so on. For example:

\[ \rightarrow \text{(generate 'modified-noun)} \]

BIG BLUE BOOK

nil

\[ \rightarrow \text{(generate 'sentence)} \]

JILL KICKED THE BIG BALL .

nil

Note: Don’t worry about the return value (above, it is nil) from generate.

Finally, write a function gen which takes an integer value \( n \) and a symbol as its two arguments, and repeatedly calls on generate to generate \( n \) random realizations of the symbol as represented by the grammar. The return value of gen should be nil. For example, we can generate 4 sentences from the above grammar as follows:

\[ \rightarrow \text{(gen 4 'sentence)} \]

A BLUE BABY KISSED THE BORING BALL .
BERTRAND RUSSELL LEARNED FROM THE BIG BORING LARRY BIRD .
JILL KICKED LARRY BIRD .
WHO KICKED THE BIG BLUE BOOK ?

nil

3.5.1.4 Optional Additions

- Create some of your own grammars for use with the sentence generator.

3.5.2 Project: Animals

3.5.2.1 The Game of Animals

For this project, you will write a program which plays a variation of a game called Animals. What makes this game special is that it is a “learning program” — your program will get smarter the more you play with it.

Here is a quick dialogue that points out most of the features that will be supported by the version of Animals to be implemented in this project:
3.5. PATTERN MATCHING PROJECTS

-> (animals)

Hi there!
Please give me information or ask questions at the => prompt...

=> (a flounder is a fish)
   (OK -- I understand)
=> (a fish is an animal)
   (OK -- I understand)
=> (a fish is a swimmer)
   (OK -- I understand)
=> (what is a flounder)
   (a flounder is a fish)
=> (what is a swimmer)
   (a swimmer is something more general than a fish)
=> (is a flounder an animal)
   (yes -- a flounder is an animal)
=> (why is a flounder a swimmer)
   (because a flounder is a fish and a fish is a swimmer)
=> (why is a flounder a flounder)
   (because they are identical)
=> (why is a flounder a fish)
   (because you told me so)
=> (an animal has a heart)
   (OK -- I understand)
=> (has a fish a heart)
   (yes -- a fish has a heart)
=> (a heart is an organ)
   (OK -- I understand)
=> (has a flounder an organ)
   (yes -- a flounder has an organ)
=> (exit)
Thank you for playing Animals...

The Animals program has the ability to store and retrieve relational data and perform simple
inferences on this data. The program is conversational, supporting a variety of question-answer modes. The program works by interpreting simple statements or questions, and then invoking functions to manipulate or reference its relational knowledge (stored as some type of knowledge network). For instance, from the above input, something like the following knowledge network should be built up:

Note: The actual network that you will build for this project will probably be slightly more complicated than this, in the sense that more properties will have to be stored (i.e., in addition to the AKO and TRAIT properties pictured above).

Here is a summary of the statements and questions which should be handled by the program (obj1 and obj2 represent objects, like a flounder or an organ):

- **Statements**
  1. (obj1 is obj2)
     Response: Simple acknowledgement.
  2. (obj1 has obj2)
     Response: Simple acknowledgement.
  3. (exit)
3.5. PATTERN MATCHING PROJECTS

- Questions

1. (is obj1 obj2)
   Response: Inform user whether or not obj1 is a kind of obj2.

2. (what is obj1)
   Response: Returns any information that has been input about obj1 (in the form of the first type of statement) as follows:
   - If there has been input of the form (obj1 is obj2), then return all the information that has been input as such (see (what is a flounder) in above dialogue).
   - If there has been no input of the form (obj1 is obj2) but there has been input of the form (obj2 is obj1), then return all the information that has been input as such (see (what is a swimmer) in above dialogue).
   - If neither of these two cases holds, respond that you have no information on obj1.

3. (why is obj1 obj2)
   Response: If obj1 is not a type of obj2, then say so. If obj1 is a type of obj2, then return the sequence of relations which allows the conclusion that obj1 is a type of obj2 (see (why is a flounder a swimmer) in above dialogue). Note the special cases which are also implied by the above dialogue.

4. (has obj1 obj2)
   Response: Inform user whether or not obj1 possesses the trait obj2. Note that “inherited” traits should be counted (see (has a flounder an organ) in above dialogue).

If the user’s input does not fit any of these seven forms, the program should give a suitable response and ask for further input. It should not exit or crash!

In writing your program, feel free to use the pattern-matcher from the preceding section. You may also borrow code from the discussion on ako-networks. Be careful, though, because the network which must be built up for this project is a bit more complicated since we are now allowing for objects to be related to more than one other object. For instance, in the above dialogue, we have the statements (a fish is an animal) and (a fish is a swimmer). This suggests that properties should be stored as lists, rather than as single atoms (as was done with the ako-network).
Two more minor points are of interest. First, notice that the program correctly handles articles (“a” or “an”) before the names of objects. Figure out a way to store this information for each object so that you will be able to supply output which has correct articles; you may assume that the correct article is entered when an object is referred to for the first time. Second, watch out for circular relations. You may allow or disallow such relations, but whichever you choose to do, be sure that the program doesn’t get caught in any infinite loops.

### 3.5.2.2 Optional Additions

- It would be quite useful to be able to save (in a file) the information that has been offered during a session of Animals when the program is exited. That way, your program could “remember” everything it learned from its last session and can be that much smarter the next time it is run. Try to figure out a way to save the information to a file so that it can simply be loaded back in.

- Implement questions of the form (why has $obj1$ $obj2$). For instance, if (why has a flounder an organ) were typed at the end of the above dialogue, the response should be something like:

  (because a flounder is a fish and a fish is an animal and an animal has a heart and a heart is an organ)

- Allow for objects to be entered without an article (i.e., “a” or “an”). For instance, the expression (a mammal has hair) could be treated as correct input. You might find it easiest to implement this modification within the pattern-matcher itself.

- Allow for punctuation in the input so that statements will end with a period and questions with a question mark. Then, (a fish is a swimmer.) and (why is a flounder a fish?) would be acceptable input. This is not a trivial change since the punctuation must actually be stripped away from the final atom in the input list.

- Make the input (and output) totally free of parentheses.
Chapter 4

Searching Strategies

by Jason Abrevaya and Harry Lewis

4.1 Searching a Network

Searching is the process by which a solution path for a given problem is found. To be a bit more precise, let’s consider the two examples of searching problems. The first is to attain checkmate within two moves from a given configuration of a chess board; in this case, a search would look for the sequence (path) of moves that would result in checkmate. The second is to find the path out of a maze; for a given maze, a solution path found by a search might look something like (up right right down ... ). In both cases, a searching process would involve looking through the possible moves that can be made from a given state of the problem in an attempt to attain a solution to the problem.

Just as there are many searching problems, there are also many types of searching strategies which can be used to solve these problems. Each method has its advantages and disadvantages, and a large part of solving a searching problem entails choosing the appropriate method to use. In this section, two of the most basic searching strategies — depth-first search and breadth-first search — will be discussed. After understanding these two strategies, it’s not very difficult to implement other types of searching strategies, a few of which will be briefly mentioned at the end of the section.

To examine depth-first and breadth-first searches, we will use a problem which is quite general in nature. The problem involves searching a network (or graph), which is a collection of nodes connected together in pairs by edges. For example, the following network has five nodes (A, B, C, S, F) and six edges (S-A, A-F, A-B, C-F, B-C, S-B):
This network is undirected, meaning that the edges can be traversed in either direction. That is, it is acceptable to go from S to A or from A to S along the S-A edge. Not all networks, however, are undirected. To indicate that a network is directed, arrows are used to represent the edges, with the arrows pointing in the direction that the edge may be traversed:

In this example, it is acceptable to travel from X to Y, but there is no place to go from Y since no arrows emanate from node Y.

To represent networks in LISP, we will associate each node with a neighbors property, a list of all the nodes that can be reached from that node via a single edge. The representations for the two illustrated networks, then, would be:

<table>
<thead>
<tr>
<th>node</th>
<th>neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(S B F)</td>
</tr>
<tr>
<td>B</td>
<td>(S A C)</td>
</tr>
<tr>
<td>C</td>
<td>(B F)</td>
</tr>
<tr>
<td>S</td>
<td>(A B)</td>
</tr>
<tr>
<td>F</td>
<td>(A C)</td>
</tr>
<tr>
<td>X</td>
<td>(Y)</td>
</tr>
<tr>
<td>Y</td>
<td>nil</td>
</tr>
<tr>
<td>Z</td>
<td>(Y)</td>
</tr>
</tbody>
</table>

With this representation of networks, the problem we’ll consider is to determine if there is a
4.2. DEPTH-FIRST SEARCH

path (i.e., a sequence of edges) connecting one node to another and, if so, to find such a path. For example, in our first graph, some paths from S to F are (S A F), (S B C F), and (S A B C F). (There are actually an infinite number of possible paths from S to F; why?)

4.2 Depth-First Search

The paths described above from S to F can be seen by simply examining the picture of the first graph. But for larger and larger graphs, an eyeballing method is not practical. We’d like to use our representation of this network so that we can let an algorithm do the work of finding a path for us. Depth-first search, the first algorithm we’ll consider, proceeds as follows:

DFS (from START node to FINISH node):

if START = FINISH
then SUCCESS
else until SUCCESS
  if all neighbors of START have been visited
  then FAILURE
  else do DFS from (an unvisited neighbor of START) to FINISH

In this algorithm, a “visited” neighbor is one that has already been encountered on the current search path. To clarify things, let’s take a look at the the order in which paths would be explored by depth-first search in our undirected graph above:

S
S A
S A S  (failure -- S already visited;
        look at next neighbor of A)
S A B
S A B S  (failure -- S already visited;
        look at next neighbor of B)
S A B A  (failure -- A already visited;
        look at next neighbor of B)
S A B C
S A B C B  (failure -- B already visited;
            look at next neighbor of C)
S A B C F  (success -- final node F has been reached!)

The path (S A B C F) is the first solution found by the algorithm. If we decided not to stop the algorithm upon its first success, we could actually find all paths connecting S to F which
contain no node more than once. For this problem, however, a single solution is sufficient, and our implementation of the algorithm will cease its operation upon the first success.

Here is a diagram showing the order in which the edges of our undirected graph are traversed during depth-first search. The dotted arrows represent the traversals which resulted in failure. The solid arrows (labeled 1, 3, 5, 7) constitute the solution path:

To implement depth-first search in LISP, we represent each path by the list of nodes contained in that path (in the order that they are visited). We keep a list of all paths that remain to be explored (this would be ((S)) at the start). Then, we repeatedly remove the first path from the list (if there are no paths left in the list, then the search has no solution). If the removed path has the final node as its last element, then a solution has been found. If not, replace this path, at the beginning of the list of unexplored paths, by all one-edge extensions of the path to nodes not already on the path. To see how this works, we revisit our example from above:

Begin with a single path containing the start node:

((S))

Replace (S) by extensions to S’s unvisited neighbors, A and B:

((S A) (S B))

Replace (S A) by extensions to A’s unvisited neighbors, B and F:

((S A B) (S A F) (S B))

Replace (S A B) by the extension to B’s unvisited neighbor, C:

((S A B C) (S A F) (S B))
4.2. DEPTH-FIRST SEARCH

Replace (S A B C) by the extension to C’s unvisited neighbor, F:

((S A B C F) (S A F) (S B))

F is the final node of the first path; (S A B C F) is a solution!

The type of data structure which we’re using here is called a stack, and can be thought of just like a stack of plates or a stack of books. You always place an item on top of the stack and always remove an item from the top as well. A stack is often referred to as a last-in-first-out (or LIFO) data structure since the most recent item placed on a stack is the first to be taken off. In our implementation, the beginning of the list of paths can be considered the top of a stack. At each iteration, the path on top of the stack is removed (or popped) and its extensions are placed (or pushed) right back on top of the stack.

Here is the LISP code for depth-first-search:

```
(defun depth-first-search (start finish)
  (depth-explore (make-path-list start) finish))
```

```
(defun depth-explore (pending finish)
  (cond ((empty-list pending) nil)
        ((equal finish (end-of (first-of pending)))
          (solution-form (first-of pending)))
        (t (depth-explore (combine (expand (first-of pending))
                                  (rest-of pending))
           finish))))
```
CHAPTER 4. SEARCHING STRATEGIES

; expand, given a path, returns a list of paths obtained by
; extending it in all possible ways to nodes which are not
; themselves already on the path and are reachable from the
; last node on the path
(defun expand (path)
  (apply #'append
    (mapcar (extend-path-fn path)
      (neighbors (end-of path)))))

; extend-path-fn returns a function(!) which takes a node a
; returns a list of 0 or 1 path. This function returns a list
; of no paths if the node is on the path already. Otherwise,
; extend-path-fn returns a list of the single path obtained by
; adding that node to the end of the path.
(defun extend-path-fn (path)
  (function (lambda (node)
      (if (on-path node path)
        nil
        (list (add-to-end node path))))))

; Representation-dependent code

; The nodes of the network are atoms, and each atom has as its
; "neighbors" property a list of the nodes that can be reached
; by traversing a single edge. This representation has the
; disadvantage that each node must be an atom and that each
; node can belong to only one network (unless additional
; properties are used).

; Here are the property values to define a simple network:
(putprop 'S '(A B) 'neighbors)
(putprop 'A '(S B F) 'neighbors)
(putprop 'B '(S A C) 'neighbors)
(putprop 'C '(B F) 'neighbors)
(putprop 'F '(A C) 'neighbors)
4.2. DEPTH-FIRST SEARCH

; make-path-list creates a set of paths consisting of a single
; path which contains a single node. This is used for
; initializing the search process.
(defun make-path-list (node)
  (list (list node)))

; solution-form returns path in a form that is acceptable as a
; solution (i.e., so that the nodes appear in order, from start
; to finish)
(defun solution-form (path)
  path)

; empty-list returns t iff lst has no paths
(defun empty-list (lst)
  (null lst))

; first-of returns the first path in list-of-paths
(defun first-of (list-of-paths)
  (car list-of-paths))

; rest-of returns list-of-paths without the first path
(defun rest-of (list-of-paths)
  (cdr list-of-paths))

; end-of returns the last node on a path
(defun end-of (path)
  (cond ((= 1 (length path)) (car path))
    (t (end-of (cdr path)))))

; add-to-end puts a node at the end of a path
(defun add-to-end (node path)
  (append path (list node)))

; on-path returns t iff node is on path
(defun on-path (node path)
  (member node path))
CHAPTER 4. SEARCHING STRATEGIES

; combine makes a single list of paths from two existing lists
; of paths
(defun combine (list-of-paths-1 list-of-paths-2)
  (append list-of-paths-1 list-of-paths-2))

; neighbors returns a list of the neighbors that a node has
(defun neighbors (node)
  (get node 'neighbors))

This code is broken up into two distinct sections: representation-independent code and representation-
dependent code. Notice that none of the code in the representation-independent section refers to the
representation which we have chosen for networks or paths. The functions in the representation-
dependent section perform all the necessary manipulations and references required by the chosen
representation. The reason for this modular approach is that we can entirely change our representa-
tion of networks and/or paths (either for this problem or a different problem) without having to
touch the representation-independent code at all. For instance, it's probably more efficient to store
paths in reverse order since we can reference the last node of a path by just taking the car of a list
within the function end-of. In order to support this new representation of paths, only the functions
solution-form, end-of, and add-to-end would have to be altered.

4.3 Breadth-First Search

One thing to notice about depth-first search is that no care is given to the length of the solution path.
When applied to our example, depth-first search found a solution path containing five nodes even
though there existed solution paths which contain fewer nodes. In contrast, breadth-first search
prioritizes paths by the number of nodes in the path. That is, breadth-first search looks first for
a solution among paths of length one, then among paths of length two, and so on. As a result,
breadth-first search invariably finds the solution path containing the minimal number of nodes.

In our example, the graph would be explored by breadth-first search as follows:
S      (no solutions of length 1; extend (S) by one node)
S A
S B      (no solutions of length 2; extend (S A) by one node)
S A S    (failure -- S already visited)
S A B
S A F    (success -- final node F has been reached!)

To implement breadth-first search in LISP, it turns out that we need only make a minor modi-
fication to our depth-first search implementation. Rather than placing the extensions from the next
4.3. BREADTH-FIRST SEARCH

unexplored path at the beginning of the list, we will place them at the end of the list so as to keep the shortest paths at the beginning of the list:

Begin with a single path containing the start node:

((S))

Remove (S) and append extensions to S’s unvisited neighbors, A and B:

((S A) (S B))

Remove (S A) and append extensions to A’s unvisited neighbors, B and F:

((S B) (S A B) (S A F))

Remove (S B) and append extensions to B’s unvisited neighbors, A and C:

((S A B) (S A F) (S B A) (S B C))

Remove (S A B) and append the extension to B’s unvisited neighbor, C:

((S A F) (S B A) (S B C) (S A B C))

F is the final node of the first path; (S A F) is a solution!

By placing new items onto the end of the list, we are now using a data structure commonly referred to as a queue. A queue, like a queue at a bank or a cafeteria, has the property that items are always removed from the head of the queue and new items are always added to the tail of the queue, making it a first-in-first-out (or FIFO) data structure. In our representation, the beginning of the list can be thought as the head of a queue and the end of the list as the tail.

Since the implementation hasn’t changed much from depth-first search, we’re able to use most of the functions from the code for depth-first search:

; breadth-first-search takes a start and finish node and returns ; a path connecting them as found by the breadth-first search ; algorithm. If no path is found, the return value is nil.
(defun breadth-first-search (start finish)
  (breadth-explore (make-path-list start) finish))
CHAPTER 4. SEARCHING STRATEGIES

; breadth-explore takes a pending list of paths and a finish
; node and uses breadth-first search to explore new paths until
; finding a solution path
(defun breadth-explore (pending finish)
  (cond ((empty-list pending) nil)
        ((equal finish (end-of (first-of pending)))
         (solution-form (first-of pending)))
        (t (breadth-explore
            (combine (rest-of pending)
                     (expand (first-of pending)))
            finish))))

In fact, the only difference between depth-explore and breadth-explore is that the order of the arguments in the combine call is switched around. This switch parallels our switch from a stack data structure to a queue data structure — replacement paths are placed at the beginning of the list for depth-first search and at the end of the list for breadth-first search.

4.4 Tree Representation of Searches

The difference between the two strategies, if not already clear, is seen best when the network is in the form of a tree. The following diagrams depict the order, starting from the root node A, in which the nodes of a tree would be visited by the two searching strategies:
With depth-first search, the search presses as far down as possible (attaining a maximum “depth”) before backing up and going down in another direction. On the other hand, breadth-first search explores the tree by levels, completing one level of the tree before proceeding to the next level.

Even if the network being searched is not a tree, it’s quite descriptive to diagram the traversal of searches in the form of a tree. For instance, we can diagram the depth-first and breadth-first searches of our original S-A-B-C-F graph as follows:

### Depth-First Search

![Depth-First Search Diagram](image1.png)

### Breadth-First Search

![Breadth-First Search Diagram](image2.png)

## 4.5 Depth-First Search vs. Breadth-First Search

Which strategy should be used for searching problems? As you might expect, there is no clear-cut answer. Breadth-first search is certainly the preferred method when searching for solution paths of minimal length. However, when the length of the solution path is unimportant, the choice of search strategy usually boils down to the relative speed of the strategies in finding a solution path. For depth-first and breadth-first search, the time required in finding a solution depends largely on the specific problem at hand (and also on the order in which “neighbors” are examined). If it is unlikely that you will find a short solution path, depth-first search may be preferable since breadth-first will waste a lot of time searching through paths of short length. This criterion is not fail-safe, however; depth-first search may result in a lengthy “wild-goose chase” if the routes that must be searched
first are very deep (possibly infinitely deep!) and do not result in a solution path. In such a case, possible adjustments that can be made to the search strategy include switching the order in which neighbors are visited (which will hopefully make the solution path occur closer to the beginning of the search) or placing a limit on the depth of the search (which will prevent “wild-goose chases,” but may also preclude finding a solution if the limit chosen is not large enough).

4.6 State-Space Search

The only type of searching problem which has been considered to this point is searching for paths in networks. In this problem, a move from one node to another does not affect the network itself in any way. Obviously, not all problems have this characteristic; for instance, a move in a chess game alters the configuration of the pieces on the board and, as a result, changes the set of legal moves which remain for the opponent. To perform a search in a chess-related problem, then, involves searching what is commonly called a state-space. In chess, any possible configuration of the board is a state, and the state-space is the collection of all possible configurations of the board (i.e., the set of all states). The moves in the game are the instruments by which one state can be transformed to another state.

Chess is just one example of a large class of problems known as state-space search problems. Since the size of the state-space and the number of possible moves makes the implementation of a computerized chess player (a good one, at least) quite an ambitious task, we’ll confine ourselves to a more modest example of state-space search. This example, while admittedly simple, does display the characteristics held in common by all state-space searches — a set of legal states and a set of moves to change these states.

The state-space problem we’ll consider centers around the following scenario:

Three girls from an all-girls boarding school have been permitted to go on a picnic with three boys from a nearby all-boys boarding school. Both schools are located on the left bank of a river. Since the park is on the right bank of the river, the six children have been provided with a boat in order to cross the river. Unfortunately, the boat can only hold a maximum of two people at any one time, and the headmistress from the all-girls school has mandated that the girls can’t be left outnumbered by the boys on either bank of the river (for fear that the teasing would become unbearable).

The objective is to get all six children from the left bank of the river to the right bank of the river so that they can enjoy their picnic at the park.

The states in this problem are the possible arrangements of the boys, the girls, and the boat on the two banks of the river. A move consists of one or two of the children traveling in the boat
across the river. The goal state in this problem is the state in which all six children (and the boat) are on the right bank of the river. To represent this in a diagram, we can consider each state as a node in a graph with the edges representing possible boat trips. For simplicity, only the state of the left bank is shown (since the objects on the right bank are fully determined by what’s on the left bank):

This graph is undirected since any move is reversible (i.e., the boat can simply return with the same passengers to the bank from which it came).

In our implementation, each state will be represented by a triple which describes the objects on the left bank of the river. The first number in the triple is the number of boats, the second the number of boys, and the third the number of girls. For instance, the triple (1 2 1) represents the state in which the boat, two boys, and a girl are on the left bank.

We’ll use a depth-first search strategy to explore the state-space, maintaining a list *states-seen*
in order to avoid an infinite loop in our search:

; Representation-independent code - nothing in this section
; makes any assumption about how states and moves are
; represented
;
; the main routine picnic initializes the *states-seen* list and
; invokes the searching process
(defun picnic ()
  (setq *states-seen* (list *initial-state*))
  (initial-monologue)
  (if (catch 'try-all-moves (try-all-moves *initial-state*))
      (report-success) ; come back here if a "throw" occurs
      (report-failure))) ; come back here if no "throw" occurs

; try-all-moves goes through all the possible moves, trying
; to apply each one to state
(defun try-all-moves (state)
  (do ((movelst *possible-moves* (cdr movelst)))
      ((null movelst))
    (try-a-move (car movelst) state)))

; try-a-move attempts to extend the current path by one move,
; and, if this can be done, recursively tries all extensions
; of that path. If the goal state is reached, there is a return
; to the top-level. The path is represented implicitly by the
; history of recursive calls to try-a-move (that is, by the
; stack of function calls).
(defun try-a-move (move state)
  (cond ((applicable? move state)
      (let ((new-state (apply-move move state)))
        (report-move move)
        (cond ((bad-state? new-state)
              (report-bad-state new-state))
              ((seen-state-before? new-state)
               (report-old-state)))
       ))
(t
  (report-state new-state)
  (if (success? new-state) ; jump out of
    (throw 'try-all-moves t)) ; recursion
  (record-new-state new-state)
  (try-all-moves new-state)
  (report-backing-up state))))))) ; if we get
; this far,
; no extension
; of the
; current path
; succeeded

; applicable? returns t iff the boat can legitimately be
; carrying its passengers given the bank from which the boat
; originated
(defun applicable? (move state)
  (and (<= (number-of-girls move)
     (number-of-girls (from-side state)))
   (<= (number-of-boys move)
     (number-of-boys (from-side state)))))

; bad-state? returns t iff boys outnumber girls on either bank
(defun bad-state? (state)
  (or (> (number-of-boys (left-bank state))
        (number-of-girls (left-bank state))
        0)
      (> (number-of-boys (right-bank state))
          (number-of-girls (right-bank state))
          0)))

; Representation-dependent code
;
;; seen-state-before? returns t iff state has been seen already
(defun seen-state-before? (state)
  (member state *states-seen* :test #'equal))
;; the optional "equal" parameter overrides the use of "eq" as
;; the default predicate used by "member" (necessary since
;; states are not atoms)

;; apply-move returns the state which results from applying move
;; to state
(defun apply-move (move state)
  (if (on-left-bank state)
      (pairwise-subtract state move)
      (pairwise-add state move)))

;; success? returns t iff no one remains on the left bank
(defun success? (state)
  (= 0 (number-of-boys state) (number-of-girls state)))

;; record-new-state adds state to the list of states already seen
(defun record-new-state (state)
  (setq *states-seen* (cons state *states-seen*)))

;; number-of-boys returns the number of boys on a bank
(defun number-of-boys (bank)
  (cadr bank))

;; number-of-girls returns the number of girls on a bank
(defun number-of-girls (bank)
  (caddr bank))

;; from-side returns the triple representing the bank on which
;; the boat is
(defun from-side (state)
  (if (on-left-bank state)
      (left-bank state)
      (right-bank state)))
4.6. STATE-SPACE SEARCH

; on-left-bank returns t iff the boat is on the left bank
(defun on-left-bank (state)
  (= 1 (car state)))

; left-bank returns the triple representing the left bank
(defun left-bank (state)
  state)

; right-bank returns the triple representing the right bank
(defun right-bank (state)
  (complement state))

; complement returns the triple which represents the state of
; the other bank
(defun complement (bank)
  (list (- 1 (car bank)) ; only one boat
        (- 3 (cadr bank)) ; only three boys in all
        (- 3 (caddr bank)))) ; only three girls in all

; pairwise-subtract returns a list containing the results
; obtained by subtracting the elements of a and b in pairs
; (e.g., if a is (1 2 1) and b is (1 1 1), the result is
; (0 1 0))
(defun pairwise-subtract (a b)
  (pairwise-op #'-' a b))

; pairwise-add returns a list containing the results obtained
; by adding the elements of a an b in pairs
(defun pairwise-add (a b)
  (pairwise-op #'+ a b))

; pairwise-op performs the operation fn on the elements of a
; and b, in pairs, and returns a list of the results
(defun pairwise-op (fn a b)
  (mapcar fn a b))
; Initial values of global variables
;

; *possible-moves* is the set of all triples which represent possible boat trips (e.g., (1 1 1) represents a boat carrying one boy and one girl)
(setq *possible-moves* '((1 1 1) (1 1 0) (1 0 1) (1 0 2) (1 2 0)))

; *initial-state* is a boat, three boys, and three girls on the left bank
(setq *initial-state* '(1 3 3))

; Print routines
;

; initial-monologue introduces the user to the program
(defun initial-monologue ()
  (princ "Welcome to the Picnic Problem.")
  (terpri)
  (princ "In a triple (a b c), ")
  (terpri)
  (princ " a is the number of boats")
  (terpri)
  (princ " b is the number of boys")
  (terpri)
  (princ " c is the number of girls")
  (terpri)
  (report-state *initial-state*))

; report-failure reports that no solution could be found
(defun report-failure ()
  (princ "No solution found...")
  (terpri))
4.6. STATE-SPACE SEARCH

; report-success reports that a solution has been found
(defun report-success ()
    (princ "Problem solved!"
    (terpri))

; report-bad-state reports that girls have been outnumbered on
; a bank
(defun report-bad-state (state)
    (terpri)
    (princ "*** Left bank: ")
    (princ (left-bank state))
    (terpri)
    (princ "*** Right bank: ")
    (princ (right-bank state))
    (terpri)
    (princ "*** Girls are outnumbered! Forget that move.")
    (terpri))

; report-old-state reports that a state has been revisited
(defun report-old-state ()
    (princ "A previous state. Forget that move.")
    (terpri))

; report-backing-up reports that the search is backing up to try
; another path
(defun report-backing-up (state)
    (princ ">>> Backing up to state ... <<<")
    (report-state state))
; report-state reports the population of the left and right banks
(defun report-state (state)
  (terpri)
  (princ "Left bank: ")
  (princ (left-bank state))
  (terpri)
  (princ "Right bank: ")
  (princ (right-bank state))
  (terpri))

; report-move reports the move that is being applied to obtain a new state
(defun report-move (move)
  (princ "Apply move ")
  (princ move)
  (princ " resulting in "))

This code differs from our earlier implementation of depth-first search in two major ways. First, we do not keep track of the paths which remain unexplored in an actual stack. Rather, as each path extension is explored, there is a message printed depending on the nature of the resulting state. Instead of using an actual stack, we are implicitly relying on the stack of recursive function calls to make sure that all unexplored paths are eventually explored until a success occurs. The second major difference is that our recursion is abruptly ended (via the catch-throw mechanism) when a success occurs. The use of this mechanism allows us to cut to the chase once a solution is found rather than passing a t value up through many levels of recursion until it reaches the top-level.

A logical question to ask is whether we could have used the original code which we wrote for depth-first search and breadth-first search and apply it to this problem. The answer is yes. And, due to the modular approach which was adopted, only the representation-dependent code must be altered. For this problem, there is no use of properties; instead, the neighbors function has to use the variable *possible-moves* and some of the functions from the above code to determine the neighbors of any given state. The only other function which needs to be changed is on-path (so that it uses the predicate equal). Here are the necessary changes and the resulting output from the depth-first-search and breadth-first-search:
4.7. OTHER SEARCHING STRATEGIES

; neighbors returns a list of the legitimate states which can
; be reached from node by a single move
(defun neighbors (node)
  (apply #'append
    (mapcar (function (lambda (state)
        (if (bad-state? state) nil ; don’t include bad states
        (list state))))
    (extensions node))))

; extensions returns a list of all states (even ones in which
; girls are outnumbered) which can be reached from node by a
; single move
(defun extensions (node)
  (apply #'append
    (mapcar (#'(lambda (move)
        (if (applicable? move node)
        (list (apply-move move node))
        nil))) ; don’t apply illegal moves
    *possible-moves*)))

; on-path returns t iff node is on path
(defun on-path (node path)
  (member node path :test #'equal))

-> (depth-first-search '(1 3 3) '(0 0 0))
((1 3 3) (0 2 2) (1 2 3) (0 0 3) (1 1 3) (0 1 1) (1 2 2) (0 2 0)
 (1 3 0) (0 1 0) (1 2 0) (0 0 0))

-> (breadth-first-search '(1 3 3) '(0 0 0))
((1 3 3) (0 2 2) (1 2 3) (0 0 3) (1 1 3) (0 1 1) (1 2 2) (0 2 0)
 (1 3 0) (0 1 0) (1 2 0) (0 0 0))

4.7 Other Searching Strategies

Depth-first search and breadth-first search are sometimes called blind searches since no information about the problems being solved is used in order to direct the search. It turns out, however, that
we can improve the efficiency of a search if we utilize the knowledge that we have about a certain problem. In particular, we can use a heuristic function to determine the “value” of any given state in the problem.

Consider the example of a maze whose form is a $5 \times 5$ grid, with the entrance at location $(1, 1)$ and the exit at location $(5, 5)$. The possible moves in a maze from a given grid location (state) are up, down, left, and right. Certain locations in the grid will represent the walls of the maze; these locations can not be entered. Here is one possible $5 \times 5$ maze (blackened squares represent walls):

![Maze Diagram]

Intuitively, it seems that it is better to give high priority to searching those paths whose final nodes are closest to the exit (i.e., location $(5, 5)$). Unfortunately, depth-first and breadth-first search do not take this problem-specific information into account. To make a more intelligent searching strategy, we can create a heuristic function that evaluates how close a given path is to the exit. Specifically, we negate the sum of the horizontal distance from the exit and the vertical distance from the exit. For instance, a path ending at $(3, 4)$ would have a value of $-3$; the highest possible heuristic value would be $0$, for a path which ends at the exit location (i.e., a solution path). The negation is necessary since heuristics, by convention, place a higher worth on preferred states.

Once a suitable heuristic has been chosen, many different search techniques may be used. We describe three such techniques below:

* **Hill-Climbing Search**

  - Begin with a stack having a single path (containing the start node).
• Then, repeat the following step until the stack is empty (no solution) or the first path on the stack has the goal node as its final element (success!):
  – Remove the first path from the stack and replace it (on top of the stack) with all possible extensions, sorted by the heuristic so that the most worthwhile extensions will be explored first.

**Best-First Search**

• Same as hill-climbing, except that the entire list of paths (not only the possible extensions) are sorted according to the heuristic.

**Beam Search**

• Same as hill-climbing, except that only a set number (say, 3 or 4) of extensions are used to replace the removed path.

NOTE: Beam search is different from depth-first, breath-first, best-first, and hill-climbing since there is no guarantee that a solution will be found by beam search even if it exists. Cutting down on the number of extensions can certainly speed up the searching process, but there is a chance that the solution path might be eliminated in doing so.

To see if and how these new searching strategies improve the efficiency of our search, let’s examine how each of the searching strategies mentioned in this section performs on the maze pictured above. There are four possible moves from any location in the grid. Of course, not all are applicable in each location due to the existence of walls and boundaries, but we know that there can be at most four possible moves from any given location. As a convention, let’s say that these moves have a priority ordering of **LEFT, RIGHT, UP, DOWN**. That is, all other things being equal, a search would choose to move **LEFT** over moving **RIGHT**, move **RIGHT** over moving **UP**, and move **UP** over moving **DOWN**. It’s important to set up such a priority ordering so that we can compare the performance of our different strategies. After all, there will be times when two grid locations have the same heuristic value, and we’ll need some way of deciding which location to search first. The priority ordering offers us such a decision tool.
Let’s first examine how depth-first and breadth-first searches perform in our maze example. Here are tree diagrams which illustrate how the grid locations are visited by the two search techniques:

The inefficiency of both strategies is readily apparent — both visit each of the possible 18 grid locations (i.e., those locations not blackened in our picture) before finally reaching the maze’s exit.

Using the heuristic developed above, the efficiency of these strategies can be improved upon. Here are tree diagrams which illustrate how the grid locations are visited by a hill-climbing search and a best-first search based on our heuristic function:
The hill-climbing technique visits 14 of the 18 possible grid locations, while the best-first technique visits 13 of the 18 possible locations. Notice that the only difference between the two techniques is that hill-climbing searches the location \( (5, 1) \). Best-first doesn’t visit \( (5, 1) \) since it sees that location \( (4, 3) \) is more worthwhile (it has a heuristic value of \(-3\) as opposed to \( (5, 1) \)’s heuristic value of \(-4\)). The reader is encouraged to go through the stack operations for both of these techniques to see how they differ slightly.

The reason that we haven’t shown a diagram for beam search is that it performs identically to hill-climbing search for this particular maze as long as the size of the beam is chosen to be greater than one. If the size of the beam is chosen to be one, beam search would not come up with a solution path; it would visit three grid locations \( (1, 1), (2, 1), (3, 1) \) and cease operation without reaching the exit.
4.8 Search Strategy Projects

4.8.1 The Eight-Puzzle

The *eight-puzzle* consists of eight numbered, movable tiles set in a $3 \times 3$ frame. One cell in the frame is always empty, thus making it possible to move an adjacent numbered tile into the empty cell. The goal of the eight-puzzle is to transform some initial configuration of the puzzle into the goal configuration through a sequence of tile moves, such as “move tile #6 down, move tile #4 left, ...”:

<table>
<thead>
<tr>
<th>2</th>
<th>8</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Finding a solution to the eight-puzzle is really just another state-space search problem. Each complete $3 \times 3$ tile configuration of the eight-puzzle is a state (with the goal-state pictured above). A move of a tile serves to transform the puzzle from one state to another.

At first glance, it might seem that there are 32 possible moves for any configuration, corresponding to: “move tile #1 left,” “move tile #1 right,” “move tile #1 up,” “move tile #1 down,” “move tile #2 left,” and so on. Of course, most of these rules are not applicable from a given puzzle configuration. In fact, there can be at most four possible moves from any puzzle configuration since any move can be thought of as a movement of the empty tile! That is, whenever a tile is moved, the empty tile will either move left, right, up, or down.

The same sort of depth-first search as was used in the “Picnic Problem” can be used to solve the eight-puzzle, with two major adjustments to the strategy:

1. Since there are an extremely large number of possible states for the eight-puzzle, a limit (like 8 or 9) should be placed on the depth of any given search path. That is, if the size of a search path reaches this limit without achieving a solution, backtracking should occur. With certain initial puzzle configurations, this limit will result in no solution being found; in such cases,
the limit would have to be increased (the cost being a large increase in running time).

2. While quite helpful for debugging, messages which report backtracking and bad moves result in quite convoluted output. The final version of your program should print only those moves that are required to achieve the goal state of the puzzle. If you keep track of the states traversed in the current search path, it shouldn’t be difficult to reconstruct the solution path. For example, the program’s output might look like:

   Move tile #6 down
   Move tile #1 right
   Move tile #7 up
   etc.
   etc.

The use of a nine-element list is suggested for the representation of a puzzle’s state, where the first three elements are the tiles in the first row, the next three are the tiles in the second row, and the last three are the tiles in the third row. For instance, the list \((1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ \text{blank})\) would represent the eight-puzzle’s goal state. Your main function, solve-puzzle, should take the representation of an initial puzzle configuration as its only argument.

Try to make your code easily generalizable to the 16-puzzle \((4 \times 4\) grid), 25-puzzle \((5 \times 5\) grid), etc. Ideally, it should be possible to change the value of a single global variable in order to have the program function in the same manner for any \(n\)-puzzle where \(n\) is the square of an integer.

### 4.8.1.1 Optional Additions

- Have your program output pictures of the puzzle configurations which occur along the solution path by writing a function `print-puzzle` which takes a list representation of the puzzle. For instance, the goal state might look something like:

```
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 |   |
-------------
```

- Instead of blindly applying moves to a state, it might be more fruitful to incorporate some hill-climbing or best-first techniques to the search strategy. A possible heuristic would be a function that takes the negative of the number of numbered tiles in a given state that are out
of place (in relation to the goal state). For instance, the value of this function for the pictured
initial state would be -6; the value for the goal state is obviously 0. Test out the change
in performance using hill-climbing and/or best-first search with this heuristic. (The built-in
LISP function sort should come in handy.) Can you come up with any more intelligent
heuristic functions?

• There is a subtle problem associated with the use of a depth limit. There are cases when
this limit might actually cause the search to miss a solution whose number of moves is
less than the limit. This problem arises since we keep a list of *states-seen* without
recording at what depth a given state was seen. For instance, it might be the case that we
encounter at the limit depth a board which is one move away from the goal configuration.
Since we’ve reached the limit, backtracking will occur and this board will be recorded as
a seen state. If this board is re-encountered at a depth less than the limit, however, we
won’t attempt any moves since it’s already in *states-seen* (even though a single move
would result in solution). Try to avoid this problem by associating with each encountered
state the shallowest depth at which the state was seen; using this information, alter the search
routine so that the depth-first search will always find a solution if there is one having fewer
moves than the search limit.

### 4.8.2 Shortest Paths in Graphs

#### 4.8.2.1 Weighted Graphs

The graphs discussed in the preceding section had no weights associated with their edges. That is,
there was no sense of “how far one node is from another,” “how costly it is to travel from one node
to another,” etc. Many real-life graphs, however, do have weights associated with them. A road
map, for instance, might have distances labeling the edges connecting cities. An airline map might
instead have monetary costs associated with edges, corresponding to the cost of flights between
cities.

Here is a weighted graph representing a European road map (with mileage labels on each edge):
As with unweighted graphs, we could use a strategy like breadth-first or depth-first search in order to search for paths connecting two nodes (cities). But the addition of weights to the graph leads to the more difficult problem of finding the shortest such path (i.e., the path in which the sum of edge costs is smallest). Finding the shortest path between two nodes requires more than a simple breadth-first or depth-first search. One can imagine a version of either of these searches modified to find all paths and select the shortest, but this approach could prove extremely inefficient. The maximum number of possible paths among $N$ cities could be as large as $N!$ ($N$ factorial) — which quickly becomes computationally intractable as $N$ grows large. There are cases, however, in which an appropriately chosen heuristic results in a search strategy that cuts down tremendously on the search time.

4.8.2.2 A* Algorithm for a Road Map

For this project, you will apply a searching strategy known as the A* algorithm to the problem of finding the shortest route between two cities (according to the mileage on a road map). Given the cities Lisbon and Vienna, for example, you would want to find the road route of the least possible mileage connecting the two cities. The A* algorithm is actually just a special kind of best-first search, a strategy discussed in the preceding section. The algorithm guarantees that the
first solution which it finds will be the shortest. What makes the A* algorithm special is the heuristic that is used for the best-first strategy. An A*-algorithm heuristic must always return a value which is no greater than the total cost of the solution path (in this instance, the total distance along the shortest path).

This restriction on the heuristic seems to be a prohibitive one; after all, there is no way to know at the beginning of the search what the total distance along the solution path will be. However, for the road-map problem, it’s actually pretty easy to come up with a heuristic that qualifies as an A*-algorithm heuristic. To do so, we need some additional information — specifically, the map coordinates of each of the cities. From these coordinates, we can compute the air distance that separates any two cities. The air distance between two given cities can never be larger than the road distance between those same two cities; remember, the shortest distance between any two points is a straight line!

Before specifying the heuristic that will be used, let’s first intuitively approach the problem of finding a shortest path (using the above road-map diagram). For example, given the choice of driving from Paris to Naples via Genoa or driving from Paris to Naples via Brussels, it seems that one should go through Genoa rather than Brussels. Why? Because Genoa is much closer on the map to the goal city (i.e., much closer in terms of air distance) than is Brussels. It’s possible, however, that the priority placed upon a path through Genoa might not be warranted; for instance, it may be the case that only very indirect and/or circuitous road routes connect Genoa with Naples. In general, though, a path through Genoa to Naples will probably be shorter than one through Brussels.

Let’s translate this reasoning into a mathematical heuristic. Given the road distances from Paris to Genoa and from Paris to Brussels, the path through Genoa was preferable because the quantity

\[
\text{road-dist } (\text{Paris, Genoa}) + \text{air-dist } (\text{Genoa, Naples})
\]

is smaller than the quantity

\[
\text{road-dist } (\text{Paris, Brussels}) + \text{air-dist } (\text{Brussels, Naples}).
\]

In general, our heuristic will take an arbitrary path (beginning with the start city and ending at some intermediate city) and return the sum of the road distance accumulated so far and the air distance between the intermediate city and the goal city. To calculate air distance, the geometric formula for the distance between two points is used:

\[
\text{dist. from } (x_1, y_1) \text{ to } (x_2, y_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}
\]

Notice that this heuristic satisfies our restriction since the air distance between the intermediate city and the goal city will never be greater than the road distance along any path connecting the intermediate city and goal city.
4.8. SEARCH STRATEGY PROJECTS

4.8.2.3 Implementing the A* Algorithm

Using air distance to (under)estimate the road distance between two cities, you can implement the A* algorithm by following the best-first-search algorithm:

1. Initialize the list of partial paths to the path containing just the start city. Call this list \( L \).

2. Remove the first path, \( P \), from \( L \). If the last city in \( P \) is the goal city, then return \( P \) and exit.

3. If not, create all the new partial paths possible by extending the last city in \( P \) by one. That is, create a new partial path for each neighbor of the last city in \( P \). (You should not add a neighbor if it is already in \( P \).)

4. Merge these new partial paths into \( L \), sorting the entire list \( L \) by our heuristic — total accumulated cost (road distance) plus the estimated cost (air distance) to the goal city.

5. If there are no paths left in \( L \), return \( \text{nil} \) to indicate that no path connects the two cities. If \( L \) still contains paths, go back to step 2.

You must choose an appropriate structure to represent each partial path and the cost associated with it. The easiest way to represent a path between two cities is to have a list containing all the cities in the path. Remember that for each partial path you will need to know both the accumulated cost so far and the estimated cost to the goal. This suggests choosing a structure which contains both the partial path and these two costs.

In order to represent the cities of the graph, you may assume the existence of two global variables — *position-list* and *neighbors-list*. The list *position-list* will be composed of sub-lists, each containing a city name along with the city’s x-coordinate and y-coordinate. The list *neighbors-list* will also be composed of sub-lists; each sub-list will contain a city’s name and then a list which contains each of its immediate neighbors and the road mileage necessary to reach these neighbors. To see precisely how these can be set up in LISP, consider the representation of the European road map which was pictured above:

```lisp
(defvar *position-list*
  '((Copenhagen 747 1438)
    (Hamburg 842 1277)
    (Berlin 681 1230)
    (Warsaw 369 1249)
    (Amsterdam 1041 1192)
    (Brussels 1069 1079))
```
(Prague 625 1060)
(Paris 1154 946)
(Bern 927 889)
(Munich 738 908)
(Vienna 520 946)
(Budapest 397 918)
(Belgrade 293 747)
(Trieste 596 785)
(Genoa 852 662)
(Rome 653 492)
(Madrid 1495 416)
(Naples 549 435)
(Lisbon 1760 322))

(defvar *neighbors-list*
   '((Copenhagen
      ((Hamburg 207)))
     (Hamburg
      ((Amsterdam 338) (Berlin 182) (Copenhagen 207)))
     (Berlin
      ((Bern 628) (Prague 219) (Warsaw 345)
       (Hamburg 182)))
     (Warsaw
      ((Prague 479) (Vienna 464) (Budapest 394)
       (Berlin 345)))
     (Amsterdam
      ((Munich 526) (Bern 558) (Brussels 164)
       (Hamburg 338)))
     (Brussels
      ((Bern 497) (Genoa 740) (Paris 225)
       (Amsterdam 164)))
     (Prague
      ((Vienna 185) (Munich 194) (Berlin 219)
       (Warsaw 479)))
     (Paris
      ((Genoa 629) (Madrid 805) (Brussels 225)))

(Bern
   ((Munich 311) (Trieste 489) (Genoa 304)
     (Madrid 1104) (Berlin 628) (Amsterdam 558)
     (Brussels 497)))
(Munich
   ((Vienna 280) (Rome 582) (Amsterdam 526)
     (Prague 194) (Bern 311)))
(Vienna
   ((Budapest 155) (Trieste 317) (Belgrade 501)
     (Warsaw 464) (Prague 185) (Munich 280)))
(Budapest
   ((Trieste 384) (Belgrade 263) (Warsaw 394)
     (Vienna 155)))
(Belgrade
   ((Trieste 403) (Vienna 501) (Budapest 263)))
(Trieste
   ((Genoa 361) (Rome 442) (Bern 489) (Vienna 317)
     (Budapest 384) (Belgrade 403)))
(Genoa
   ((Madrid 951) (Rome 328) (Brussels 740) (Paris 629)
     (Bern 304) (Trieste 361)))
(Rome
   ((Naples 134) (Munich 582) (Trieste 442)
     (Genoa 328)))
(Madrid
   ((Lisbon 339) (Paris 805) (Bern 1104) (Genoa 951)))
(Naples
   ((Rome 134)))
(Lisbon
   ((Madrid 339))))

You will probably find it useful to use these global variables in order to associate certain properties with each of the cities.

Your main function, find-shortest-path, should take two arguments — a start city and a destination city. If no solution path exists, a failure should be reported, and the value nil should be returned. If there exists a solution path, the return value of find-shortest-path should be a list containing the solution path (i.e., a list of cities) and the road distance of this solution path.
For example, using the data above, your output should look something like the following:

```lisp
-> (find-shortest-path 'Lisbon 'Detroit)
No path exists from Lisbon to Detroit.
nil

-> (find-shortest-path 'Lisbon 'Vienna)
((Lisbon Madrid Genoa Trieste Vienna) 1968)
```

### 4.8.3 Heuristic Maze Searches

#### 4.8.3.1 Representing Mazes

In this project, you will implement the three heuristic search strategies described at the end of the preceding section. You will apply these strategies to maze searching, but before doing so, we need to specify an appropriate representation of mazes in LISP. We refer to the sample maze introduced in the preceding section:

![Maze Diagram]

Each of the cells in the maze will have associated with it one of two symbols — wall or nowall. We have a list of our special symbols in order to represent a maze row. For instance, the bottom row of the maze above would be represented by the list `(nowall nowall nowall wall nowall)`. To represent an entire maze, we use a list of lists. As a convention, we’ll say that rows with lower coordinates will appear first in the representation. With this convention, our...
4.8. SEARCH STRATEGY PROJECTS

maze above has the following representation:

```
((nowall nowall nowall wall nowall)
(nowall wall wall wall nowall)
(nowall nowall nowall nowall nowall)
(wall nowall nowall nowall wall)
(nowall nowall wall nowall nowall))
```

This representation should be stored in a global variable, *maze*. To indicate the maze’s entrance and exit, we have two additional global variables — *entrance* and *exit*. Each is an ordered pair of integers (column coordinate followed by row coordinate) corresponding to the appropriate maze cell. In the maze above, *entrance* would be (1 . 1) and *exit* would be (5 . 5). (Note: Both the entrance and exit of the maze should correspond to cells having no walls.)

4.8.3.2 Applying Depth-First and Breadth-First Searches

The first part of this project is to use the representation-independent code for breadth-first and depth-first searches in order to search a maze. You should alter the representation-dependent code (without touching the representation-independent code) so that the maze representation discussed above can be used. The state of a maze search can be represented by an ordered pair specifying the maze cell currently being visited. You should prioritize moves as follows: left, right, up, down.

4.8.3.3 Implementing Heuristic Strategies

Recall the maze heuristic specified in the preceding section: negating the sum of the horizontal distance from the exit and the vertical distance from the exit. You should implement this heuristic in LISP as a function which takes as its argument a maze cell (i.e., an ordered pair) and returns an integer.

With this heuristic, you can now implement three heuristic-based searching functions — hill-climbing-search, best-first-search, and beam-search. Like depth-first-search and breadth-first-search, these functions should take both a start state and a final state as arguments and should be entirely representation-independent. In addition, the new functions should take a heuristic comparison function as an argument, with beam-search also taking an integer (to indicate the size of beam to be used) as an argument. We might invoke the searching functions as follows:

```
(hill-climbing-search *entrance* *exit* 'maze-heuristic-cmp)
(best-first-search *entrance* *exit* 'maze-heuristic-cmp)
(beam-search *entrance* *exit* 'maze-heuristic-cmp 2)
```
A “heuristic comparison function” is a function which takes two arguments (maze cells) and compares their heuristic value (i.e., by calling upon the heuristic function which you have already written).

In order to test that your new functions are working properly, you may want to trace them in order to check that they function as diagrammed in the search trees from the preceding section. Just checking that they return a correct solution path is not sufficient since each of the discussed searching strategies arrives at the same solution path for the sample $5 \times 5$ maze. Alternatively you can come up with different mazes for which different solution paths are returned by different searching strategies.

4.8.3.4 Optional Additions

- The representation for mazes can be made far more concise. Rather than using wall and nowall, we could instead associate with each cell a value of zero or one. Using this alternate cell representation, it’s possible to represent an entire row with a single integer (how?) and, thus, an entire maze by a list of integers. Modify your representation-dependent code in order to support such a representation.

- Modify the various search routines so that they count the number of distinct states visited (for the specific task of maze searching, the number of distinct maze cells). In this way, you can quite easily compare the performance of different search strategies.

- Try to come up with a heuristic that is more intelligent than the one suggested.

- Rather than receiving a list of states as a solution, it might be nice to see the actual sequence of moves (left, right, up, down) necessary to get from the start state to the final state. For instance, a solution to our sample maze might instead be output as follows:

\[((up 2) (right 3) (up 2) (right 1))\]
Chapter 5

Propositional Calculus and Theorem-Proving

by Jason Abrevaya and Harry Lewis

5.1 Propositional Variables and Boolean Operators

The basic building blocks of the propositional calculus are propositional variables, which can take on one of two values — true or false. A propositional variable is usually represented by a single letter, like $p$ or $q$, but any label can be used. The name “propositional variable” is used since such a variable is used to represent some proposition (e.g., “It is raining outside,” “Smith is the murderer,” etc.). If, say, $p$ were used to represent the proposition “It is raining outside,” the value of $p$ would indicate whether it is raining outside — true indicating rain, false indicating no rain.

By themselves, propositional variables are not all that interesting. But we can combine these variables using boolean operators in order to create more complex expressions known as propositional formulas. The four basic boolean operators which we’ll consider are \textsc{and} ($\land$), \textsc{or} ($\lor$), \textsc{not} ($\neg$), and \textsc{implies} ($\Rightarrow$), with their symbolic equivalents specified inside parentheses. \textsc{And} is a binary operator that evaluates to true if and only if both of its operands are true. Here is a truth table that summarizes the possible values of \textsc{and}:

\[
\begin{array}{c|c|c|c}
\text{AND} & p & q & p \land q \\
\hline
T & T & T & T \\
T & F & F & F \\
F & T & F & F \\
F & F & F & F \\
\end{array}
\]
The operator OR is also a binary operator, but unlike AND, it returns true if either of its operands is true:

\[
\begin{array}{ccc}
| p & q & p \lor q |\\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T &amp; T &amp; T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T &amp; F &amp; T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F &amp; T &amp; T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F &amp; F &amp; F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
\]

The operator NOT differs from AND and OR in that it is a unary operator (i.e., it operates on a single operand). NOT simply returns the logical negation of its operand:

\[
\begin{array}{cc}
| p & \sim p |\\
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T &amp; F</td>
<td></td>
</tr>
<tr>
<td>F &amp; T</td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
\]

The operator IMPLIES is a binary operator, but unlike AND and OR, the order of its operands matters. IMPLIES returns true if and only if the truth value of the second operand follows logically from the truth value of the first operand:

\[
\begin{array}{ccc}
| p & q & p \Rightarrow q |\\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T &amp; T &amp; T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T &amp; F &amp; F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F &amp; T &amp; T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F &amp; F &amp; T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
\]

A few words of explanation are in order here. Consider first the case where \( p \) is true. The formula \( p \Rightarrow q \) can only be true in the case that \( q \) is also true. The statement “The Earth is round implies that man can fly” serves as an example where \( p \) is true and \( q \) is false. This statement is certainly false; if it were true, we would be able to logically imply the truth of the second statement from the truth of the first. Now, consider the case where \( p \) is false. Notice that our truth table tells us that \( p \Rightarrow q \) is true regardless of the truth value of \( q \). What this means is that any statement follows logically from a false statement. For instance, the statements “The blue ball is red implies man can fly ” and “The blue ball is red implies man can’t fly” are both true!

Finally, consider the formula \( \sim (p \lor q) \) — where the unary \( \sim \) operator applies only to the \( p \) since \( \sim \) is highest on the order-of-operations scale (followed, in order, by \( \land, \lor, \) and \( \Rightarrow \)). This formula is actually the same as \( (p \Rightarrow q) \), which can be checked by explicitly writing out its truth table:
5.2. Theorems and Satisfiability

Given any propositional formula and an assignment of truth values to each of its constituent variables, a truth value for the formula can be computed. As an example, consider the formula \((p \Rightarrow q) \land p \Rightarrow q\), where \(p\) is true and \(q\) is false:

\[
\begin{array}{cccc}
T & F & T & T \\
T & F & F & F \\
F & T & T & T \\
F & T & F & T \\
\end{array}
\]

This equivalence is important because it allows the use of three, rather than four, binary operators to fully express the propositional calculus. That is, any IMPLIES operator that appears in a formula can be eliminated with the use of an extra NOT and OR. This will play an important role in our computer implementation which appears later in this section.

5.2 Theorems and Satisfiability

For any formula, there are various truth-assignments which can be made to its variables. In particular, since each variable can take on one of two values, the total number of different truth-assignments for a formula is \(2^n\), where \(n\) is the number of variables in the formula. In the example above, the formula has just two variables, \(p\) and \(q\), allowing for four possible truth-assignments.

It turns out that our example computes to true for any of the four possible truth-assignments (the reader is invited to verify this). As such, this formula is called a theorem (or tautology). A theorem is simply a formula that is true under all possible truth-assignments of its variables.

In contrast, a formula that is false under all possible truth-assignments is called unsatisfiable (or inconsistent). The term “unsatisfiable” is used because the formula is not satisfied (i.e., true) for any truth-assignment; a satisfiable formula is one which can satisfied by at least one truth-assignment. Theorems are just one small subset of the set of satisfiable formulas:
An example of an unsatisfiable formula is:

\[ ((p \land q) \lor (p \land r) \lor (q \land r)) \land ((\sim p \land \sim q) \lor (\sim p \land r) \lor (\sim q \land r)) \]

To see this, we could go through the eight possible truth-assignments and see that the formula is false in each instance. Such drudgery is unnecessary, however, if we write out in English what the formula says: “Two out of three of \( p, q, \) and \( r \) are true, \textbf{and} two out of three of \( p, q, \) and \( r \) are false.” This statement is certainly false in all instances since we can’t have two true values and two false values in a group of only three values!

In fact, if a given formula is a theorem, then the negation of that formula is unsatisfiable; likewise, if a given formula is unsatisfiable, its negation is a theorem. For example, we could negate the example from above in order to arrive at a theorem:

\[ \sim ((p \land q) \lor (p \land r) \lor (q \land r) ) \land ((\sim p \land \sim q) \lor (\sim p \land r) \lor (\sim q \land r))) \]

With this basic background in logic, let’s examine a simple example involving some actual propositions. We’ll have four propositional variables, each associated with a different proposition:

\[
\begin{align*}
JMS &= \text{“Jones met Smith”} \\
SM &= \text{“Smith is the murderer”} \\
MAM &= \text{“The murder happened after midnight”} \\
JL &= \text{“Jones is lying”}
\end{align*}
\]

In addition, suppose we know the following facts (i.e., the truth values of the four variables must satisfy each of the following formulas):
5.3. TRANSFORMATIONS OF FORMULAS

\[ F_1: \sim JMS \Rightarrow (SM \lor JL) \]
“If Jones didn’t meet Smith, then Smith is the murderer or Jones is lying.”

\[ F_2: \sim SM \Rightarrow (\sim JMS \land MAM) \]
“If Smith is not the murderer, then Jones didn’t meet Smith and the murder happened after midnight.”

\[ F_3: MAM \Rightarrow (SM \lor JL) \]
“If the murder happened after midnight, then Smith is the murderer or Jones is lying.”

From these facts, we’d like to know if we can deduce that Smith is the murderer. In terms of the propositional calculus, we can do so if \((F_1 \land F_2 \land F_3 \Rightarrow SM)\) is a tautology. Let’s call this formula \(F_4\). (Notice that we’ve relaxed our notation a bit here. From here on, we’ll allow AND’s and OR’s to operate on a series of more than two variables without having to use lots of parentheses; we can do this because of the associativity of both AND and OR.) In fact, there is a truth-assignment which makes \(F_4\) false — having \(JMS\) false, \(MAM\) true, \(SM\) false, and \(JL\) true. With this truth-assignment, the formulas \(F_1\), \(F_2\), and \(F_3\) are each satisfied but \(SM\) is false, which means that the formula \(F_4\) is false. Alternatively, we can say that this truth-assignment satisfies \(\sim F_4\), the negation of the formula. \(\sim F_4\) can be written out as \(\sim (F_1 \land F_2 \land F_3 \Rightarrow SM)\). This can actually be simplified to \((F_1 \land F_2 \land F_3 \land \sim SM)\) by using some identities that will be explained in the next part of this section. For now, we’ll accept this simplification on faith, and since we know that this formula is satisfiable, it follows that it’s not inconsistent to believe \(F_1\), \(F_2\), \(F_3\), and \(\sim SM\). That is, it’s possible that the three given facts are true and that Smith is not the murderer — so Smith is off the hook!

5.3 Transformations of Formulas

It turns out that any given propositional formula can be written in many different ways, each of which is equivalent. Since we’re usually interested in simplifying the appearance of a formula as much as possible, it’s helpful to consider some logical equivalences:
Double Negation

1. \( \sim \sim p \equiv p \)

Commutative Laws

2. \( p \lor q \equiv q \lor p \)
3. \( p \land q \equiv q \land p \)

Associative Laws

4. \( (p \lor q) \lor r \equiv p \lor (q \lor r) \equiv p \lor q \lor r \)
5. \( (p \land q) \land r \equiv p \land (q \land r) \equiv p \land q \land r \)

Distributive Laws

6. \( p \land (q \lor r) \equiv (p \land q) \lor (p \land r) \)
7. \( p \lor (q \land r) \equiv (p \lor q) \land (p \lor r) \)

DeMorgan’s Laws

8. \( \sim (p \lor q) \equiv \sim p \land \sim q \)
9. \( \sim (p \land q) \equiv \sim p \lor \sim q \)

If you’re not convinced by one or more of these identities, feel free to verify them with truth tables.

We’ve already seen that any formula can be expressed solely in terms of AND, OR, and NOT if we eliminate any occurrences of IMPLIES via the equivalency of \( p \Rightarrow q \) and \( \sim p \lor q \). Additional forms of simplification can be performed with the above identities. By using DeMorgan’s Laws, for instance, it’s possible to simplify a formula to one which has negations attached only to variables. To see how this is done, we revisit the murder-mystery example from above and simplify \( \sim F_4 \):

\[
\sim (F_1 \land F_2 \land F_3 \Rightarrow SM) \\
\equiv \sim (\sim (F_1 \land F_2 \land F_3) \lor SM) \quad \text{since} \quad (p \Rightarrow q) \equiv (\sim p \lor q) \\
\equiv \sim (\sim F_1 \lor \sim F_2 \lor \sim F_3 \lor SM) \quad \text{by DeMorgan’s Law (9)} \\
\equiv F_1 \land F_2 \land F_3 \land \sim SM \quad \text{by DeMorgan’s Law (8)}
\]

Thus, by eliminating any IMPLIES operators and applying DeMorgan’s Laws appropriately, we can arrive at a formula which consists of AND’s and/or OR’s of \textit{literals}. By “literal,” we simply mean either a propositional variable or the negation of a propositional variable. In the simplified
5.3. TRANSFORMATIONS OF FORMULAS

formula above, there are four literals — three of which are variables \((F_1, F_2, F_3)\) and one of which is the negation of a variable \((\sim SM)\).

Having simplified a formula to a form with only AND’s and/or OR’s of literals, we can use the distributive laws to transform the formula into either conjunctive normal form or disjunctive normal form. A formula is said to be in conjunctive normal form (CNF) if it operates with AND on one or more sub-formulas, each of which operates with OR on one or more literals. Since this definition is quite cryptic, some examples of CNF formulas are helpful:

\[
F_1 \land F_2 \land F_3 \land \sim SM
\]

(four sub-formulas, each with one literal)

\[
(p \lor q \lor r) \land q \land (\sim p \lor r)
\]

(three sub-formulas, the first having three literals, the second having one, and third having two)

\[
p \lor q \lor r
\]

(one sub-formula, which is an OR of three literals)

Disjunctive normal form is much like conjunctive normal form, except that the operators are switched around in the definition. That is, a formula is said to be in disjunctive normal form (DNF) if it operates with OR on one or more sub-formulas, each of which operates with AND on one or more literals. Here are some examples of DNF formulas:

\[
F_1 \land F_2 \land F_3 \land \sim SM
\]

(one sub-formula, which is an AND of four literals)

\[
(p \land \sim q \land r) \lor q \lor (\sim p \land r)
\]

(three sub-formulas, the first having three literals, the second having one, and third having two)

\[
p \lor \sim q \lor r
\]

(three sub-formulas, each with one literal)

Consider the following example of transforming a formula into normal form:

Original formula: \(\sim (p \land (q \Rightarrow r))\)

First, rewrite formula without the \(\Rightarrow\) operator:

\(\sim (p \land (\sim q \lor r))\)
Next, drive in negations using DeMorgan’s Laws:

\[
\begin{align*}
&\sim p \lor \sim (\sim q \lor r) & \text{by identity (9)} \\
&\sim p \lor (\sim q \land \sim r) & \text{by identity (8)} \\
&\sim p \lor (q \land \sim r) & \text{by identity (1)}
\end{align*}
\]

This is in DNF. To get CNF, apply the distributive law:

\[
(\sim p \lor q) \land (\sim p \lor \sim r) & \text{by identity (7)}
\]

### 5.4 Representing and Simplifying Formulas in LISP

There are two LISP representations of formulas which we will consider. The first is applicable to any formula, while the second is specific to formulas in normal form. Our first representation uses prefix notation, in which the boolean operators appear before their operands (rather than in-between operands, as we have been doing). The representation can be expressed by a recursive grammar as follows:

\[
\begin{align*}
\text{FMLA} & ::= \text{PROP-VAR} | (\text{not} \ \text{FMLA}) | (\text{and} \ \text{FMLA}^*) | \\
& \quad | (\text{or} \ \text{FMLA}^*) | (\text{implies} \ \text{FMLA} \ \text{FMLA}) \\
\text{PROP-VAR} & ::= \text{a LISP symbol}
\end{align*}
\]

This grammar indicates that both AND and OR can operate on more than two operands, which follows from their associative properties. In contrast, NOT can act on just one operand and IMPLIES on just two operands.

The formula \(\sim (p \land ((q \lor \sim r) \Rightarrow s))\) would be represented in prefix notation as:

\[
(\text{not} \ (\text{and} \ (p \ (\text{implies} \ (\text{or} \ q \ (\text{not} \ r)) \ s))))
\]

This representation is useful since not, and, and or are built-in LISP functions that perform the desired boolean operations. We can also define implies to perform the appropriate boolean operation:

\[
\begin{align*}
\text{defun implies} & (f1 \ f2) \\
& (\text{or} \ (\text{not} \ f1) \ f2))
\end{align*}
\]
5.4. **REPRESENTING AND SIMPLIFYING FORMULAS IN LISP**

With each of these four functions defined, we can simply issue the statement `(eval fmla)` to determine the truth-value of `fmla` for a given truth-assignment of its propositional variables. A truth-assignment can be made by binding each of the propositional variables to a value of either `t` or `nil`.

Our goal at this point is to write a program in LISP that simplifies any formula (represented in prefix notation) into a formula containing no IMPLIES operators:

```lisp
(defun reduce-to-and-or-not (f)
  (cond ((null f) nil)
        ((pvar? f) f)
        ((conjunction? f)
         (make-conjunction
          (mapcar #'reduce-to-and-or-not (args f)))))
        ((disjunction? f)
         (make-disjunction
          (mapcar #'reduce-to-and-or-not (args f)))))
        ((negation? f)
         (make-negation
          (reduce-to-and-or-not (first-arg f)))))
        ((implication? f)
         (make-disjunction
          (list (make-negation
                  (reduce-to-and-or-not (first-arg f)))
                (reduce-to-and-or-not (second-arg f)))))))
```
CHAPTER 5. PROPOSITIONAL CALCULUS AND THEOREM-PROVING

;; Representation-dependent code
;; Representation of (general) formulas ->
;; fmla ::= pvar | (not fmla) | (and fmla*) |
;; (or fmla*) | (implies fmla fmla)
;;
;; first-arg returns the first operand of a formula
(defun first-arg (formula) (cadr formula))

;; second-arg returns the second operand of a formula
(defun second-arg (formula) (caddr formula))

;; args returns the list of operands in a formula
(defun args (formula) (cdr formula))

;; test whether a formula is a prop. var., negation, conjunction, etc.
(defun pvar? (formula) (symbolp formula))

(defun negation? (formula)
  (and (listp formula) (eq (car formula) 'not)))

(defun conjunction? (formula)
  (and (listp formula) (eq (car formula) 'and)))

(defun disjunction? (formula)
  (and (listp formula) (eq (car formula) 'or)))

(defun implication? (formula)
  (and (listp formula) (eq (car formula) 'implies)))
5.4. REPRESENTING AND SIMPLIFYING FORMULAS IN LISP

(defun make-negation (formula)
  (cond ((negation? formula) (first-arg formula))
        (t (list 'not formula))))

(defun make-conjunction (s) (cons 'and s))
(defun make-disjunction (s) (cons 'or s))

In order to implement simplification to normal form, we first introduce a second representation for formulas in LISP which will make our notation a little simpler. This representation is only for formulas in normal form, and it eliminates the need to refer to the operators \texttt{and} and \texttt{or}. A CNF formula, for instance, can be represented in LISP by a list of a sub-lists, where each sub-list represents a sub-formula; each sub-list, also called a clause, is simply a list of the literals being OR’ed together in the corresponding sub-formula. Since we know exactly how the AND and OR operators are arranged in CNF, there is no need to reference these operators within our representation; implicitly, OR operates on the literals within a sub-list, and AND operates on the sub-lists in the list. Consider the three CNF examples from above; here’s how they look under our new representation:

\[
F_1 \land F_2 \land F_3 \land \lnot SM \\
((F_1) (F_2) (F_3) ((\lnot SM)))
\]

\[
(p \lor \lnot q \lor r) \land q \land (\lnot p \lor r) \\
((p (\lnot q) r) (q ((\lnot p) r))
\]

\[
p \lor \lnot q \lor r \\
((p (\lnot q) r))
\]

Note that this same representation could also be used for DNF formulas, but the operators implicit in the representation would of course be different. Since both types of normal form can be represented in this way, it’s important to keep straight exactly which type is being used in the representation.

For a given formula \( F \) which has had its \texttt{IMPLIES} operators removed, can we come up with an algorithm to get \( F \) into either CNF or DNF in our new representation? In fact we can, and the
easiest way to go about it is to look at the four possible forms that $F$ can take:
Algorithm to convert $F$ to CNF:

Case I: $F$ is a propositional variable $p$

$$\text{CNF}(F) = ((p))$$

Case II: $F$ is a negation of a formula $G$

First, we transform $G$ into DNF. Then, according to DeMorgan’s Laws, we can get $F$ into CNF by negating all the literals in DNF($G$). Consider a short example to see why this is the case:

Say:

$$\text{DNF}(G) = ((p \sim r)(\sim p \sim q \sim r)(s))$$

i.e., $G \equiv (p \land \sim r) \lor (\sim p \land q \land \sim r) \lor (s)$

Then:

$$F \equiv \sim G$$

$$\equiv (p \land \sim r) \land (\sim p \land q \land \sim r) \land (s) \quad \text{by identity (8)}$$

$$\equiv (p \lor \sim r) \land (p \lor q \lor r) \land s \quad \text{by identity (9)}$$

$$\text{CNF}(F) = ((\sim p \sim r)(p \sim q \sim r)(\sim s))$$

which is the same as DNF($G$) with all its literals negated

In general:

$$\text{CNF}(F) = \text{result of negating all the literals in DNF}(G)$$

Case III: $F$ is a conjunction (AND) of formulas $F_1, F_2, F_3, \ldots, F_n$

If we can get each of the $n$ formulas into CNF representation, $F$ can be transformed by simply collecting together in one list all of the clauses from the $n$ CNF formulas. This transformation works since each of the formulas has implicit AND operators between its clauses (sub-lists); in effect, we are just re-arranging parentheses.

$$\text{CNF}(F) = (\text{append } \text{CNF}(F_1) \text{ } \text{CNF}(F_2) \ldots \text{ } \text{CNF}(F_n))$$

Case IV: $F$ is a disjunction (OR) of formulas $F_1, F_2, F_3, \ldots, F_n$

First, consider the simple case where $F = (F_1 \lor F_2)$. If we can get $F_1$ and $F_2$ into CNF, $F$ can be transformed into CNF by using the distributive law to create all possible clauses. Consider the following example:

Say:
CHAPTER 5. PROPOSITIONAL CALCULUS AND THEOREM-PROVING

\[ \text{CNF}(F_1) = ((p_1 \sim p_2)(p_2 p_3)) \]
\[ \text{i.e.}, \quad F_1 \equiv (p_1 \lor \sim p_2) \land (p_2 \lor p_3) \]

\[ \text{CNF}(F_2) = ((q_1)(\sim q_1 q_2 \sim q_3)) \]
\[ \text{i.e.}, \quad F_2 \equiv q_1 \land (\sim q_1 \lor q_2 \lor \sim q_3) \]

Then:
\[
F \equiv F_1 \lor F_2 \\
\equiv ((p_1 \lor \sim p_2) \land (p_2 \lor p_3)) \lor (q_1 \land (\sim q_1 \lor q_2 \lor \sim q_3)) \\
\equiv (p_1 \lor \sim p_2 \lor q_1) \\
\land (p_2 \lor p_3 \lor q_1) \\
\land (p_1 \lor \sim p_2 \lor \sim q_1 \lor q_2 \lor \sim q_3) \\
\land (p_2 \lor p_3 \lor \sim q_1 \lor q_2 \lor \sim q_3) \\
\]

\[ \text{CNF}(F) = ((p_1 \sim p_2 q_1)(p_2 p_3 q_1) \\
(p_1 \sim p_2 \sim q_1 q_2 \sim q_3) \\
(p_2 p_3 \sim q_1 q_2 \sim q_3)) \]

In the general case when \( F = (F_1 \lor F_2 \lor F_3 \lor \ldots \lor F_n) \), we first consider just \( F_1 \) and \( F_2 \) and use distribution to get \( \text{CNF}(F_1 \lor F_2) \). We use this result and distribute its clauses with the clauses of \( F_3 \) to get \( \text{CNF}(F_1 \lor F_2 \lor F_3) \). This process can be continued until we arrive at \( \text{CNF}(F_1 \lor F_2 \lor F_3 \lor \ldots \lor F_n) \).

This algorithm is self-contained except for the reference to \( \text{DNF}(G) \) in the second case. As a result, we need an algorithm for \( \text{DNF} \) before our \( \text{CNF} \) algorithm can be considered complete. As it turns out, the algorithm for \( \text{DNF} \) is very similar to the \( \text{CNF} \) algorithm due to the symmetry of the \( \text{AND} \) and \( \text{OR} \) operators. Case I, of course, remains the same. Case II is the same, except that we negate the literals in the \( \text{CNF} \) of the formula. Case III becomes like Case IV, using distribute identity (6) instead of distributive identity (7). Likewise, Case IV becomes like Case III, where the \( \text{DNF} \) lists need only be appended together:
5.4. REPRESENTING AND SIMPLIFYING FORMULAS IN LISP

Algorithm to convert $F$ to DNF:

Case I: $F$ is a propositional variable $p$

$$\text{DNF}(F) = ((p))$$

Case II: $F$ is a negation of a formula $G$

$$\text{DNF}(F) = \text{result of negating all the literals in CNF}(G)$$

Case III: $F$ is a conjunction (AND) of formulas $F_1, F_2, F_3, \ldots, F_n$

$$\text{DNF}(F) = \text{result of distribution on DNF}(F_1), \text{DNF}(F_2), \ldots, \text{DNF}(F_n)$$

Case IV: $F$ is a disjunction (OR) of formulas $F_1, F_2, F_3, \ldots, F_n$

$$\text{DNF}(F) = (\text{append DNF}(F_1) \text{DNF}(F_2) \ldots \text{DNF}(F_n))$$

Building upon the code we’ve already written, here is the code which uses the above algorithms to convert a formula into normal form. The main routine make-clause-set can be called with a formula in prefix notation, and it will return the set of clauses that represents the formula’s CNF:

; Representation-independent code
; (except that clause sets are assumed to be lists of clauses,
; and clauses are lists of literals, whatever they may be)
;

; make-clause-set takes a formula in prefix notation and returns
; a set of clauses, the representation of the CNF of the formula
(defun make-clause-set (formula)
  (normalize-clause-set
   (make-cnf (reduce-to-and-or-not formula))))

; turn a formula with "and"s, "or"s and "not"s into conjunctive
; normal form or disjunctive normal form
(defun make-cnf (formula)
  (make-nf #'make-cnf #'make-dnf #'conjunction? #'disjunction? formula))

(defun make-dnf (formula)
  (make-nf #'make-dnf #'make-cnf #'disjunction? #'conjunction? formula))
; make-nf transforms formula to cnf or dnf, depending on its
; functional parameters. make-this-nf and make-other-nf should
; be make-cnf and make-dnf for cnf, or vice versa for dnf.
; this-test? and other-test? should be conjunction? and
; disjunction? for cnf, or vice-versa for dnf
(defun make-nf (make-this-nf make-other-nf
    this-test? other-test? formula)
  (cond ((pvar? formula)
    (list (list formula)))
    ((negation? formula)
      (mapcar #'negate-all-literals
        (funcall make-other-nf (first-arg formula))))
    ((funcall this-test? formula)
      (apply #'append
        (mapcar make-this-nf (args formula))))
    ((funcall other-test? formula)
      (distribute-fully
        (mapcar make-this-nf (args formula))))))

; negate-all-literals negates all of the literals in a clause or
; a clause set
(defun negate-all-literals (s)
  (cond ((pvar? s) (make-negation s))
    ((negation? s) (first-arg s))
    (t (mapcar #'negate-all-literals s))))

; distribute-fully takes forms all possible "cross-product"
; clauses from a list of clause sets—to imitate distribution
; laws. For example, (((a) (b)) ((c) (d e)) ((f))) --->
; (((a c f) (a d e f) (b c f) (b d e f))
(defun distribute-fully (ss)
  (cond ((null ss) (list nil))
    (t (distribute-2 (car ss)
      (distribute-fully (cdr ss)))))))
5.4. REPRESENTING AND SIMPLIFYING FORMULAS IN LISP

; Representation-dependent code
;

; normalize-clause-set forms a standardized representation of
; a clause set
(defun normalize-clause-set (s)
  (eliminate-duplicates
   (sort (mapcar #'normalize-clause s) #'f-less)))

; normalize-clause forms a standardized representation
; of a clause
(defun normalize-clause (c)
  (eliminate-duplicates (sort (copy-list c) #'f-less)))

; eliminate-duplicates eliminates (adjacent) duplicates in a
; sorted(!) list
(defun eliminate-duplicates (l)
  (cond ((null l) nil)
        ((null (cdr l)) l)
        ((equal (car l) (cadr l))
         (eliminate-duplicates (cdr l)))
        (t (cons (car l)
                  (eliminate-duplicates (cdr l))))))
CHAPTER 5. PROPOSITIONAL CALCULUS AND THEOREM-PROVING

;;; f-less is a comparison function for clauses and formulas.
;;; Variables are compared alphabetically; any variable precedes
;;; any non-atomic formula; and, non-atomic formulas are compared
;;; lexicographically (i.e., as they would be printed)
(defun f-less (f1 f2)
  (cond ((null f1) (not (null f2)))
        ((null f2) nil)
        ((pvar? f1) (if (pvar? f2) (string< f1 f2) t))
        ((pvar? f2) nil)
        (t (or (f-less (car f1) (car f2))
              (and (equal (car f1) (car f2))
                   (f-less (cdr f1) (cdr f2)))))))

;;; distribute-2 takes two clause sets and returns a clause set
;;; which includes every clause that can be formed from the union
;;; of a clause in the first set and a clause in the second set.
;;; For example, (distribute-2 '((a) (b c)) '((d e) (f))) ---->
;;; ((a d e) (a f) (b c d e) (b c f))
(defun distribute-2 (s1 s2)
  (cond ((null s1) nil)
        (t (append (mapcar (prepend-clause-fn (car s1)) s2)
                 (distribute-2 (cdr s1) s2)))))

;;; prepend-clause-fn returns a function of one argument,
;;; variable-clause, which returns the union of variable-clause
;;; and fixed-clause
(defun prepend-clause-fn (fixed-clause)
  (function (lambda (variable-clause)
              (append fixed-clause variable-clause))))

In this code, there are several routines for standardizing the form of clauses and clause sets to ensure that duplicate clauses are not considered. We want to regard two clauses as identical if they have the same literals; for example, \((p \ q \ p)\), \((p \ q)\), and \((q \ p)\) are identical for our purposes. The function normalize-clause standardizes the clause representation by eliminating duplicate literals and sorting the remaining literals alphabetically in two groups — non-negated literals first and negated literals second. As long as all clauses are normalized in this way, we can check if any two clauses are equivalent by checking if they are equal.
5.5 Theorem-Proving: The Resolution Method

As we’ve just seen, any formula can be reduced to a set of clauses by the CNF construction. Each clause is a set of literals, which in CNF means the OR of those literals. How about the empty clause (i.e., the clause with no literals)? We haven’t come across the empty clause in any of our examples, but it turns out to be quite important. Actually, the empty clause is inconsistent all by itself. This point is a subtle one. Since a clause represents an OR of literals, at least one of the literals in the clause must be true in order for the clause itself to be true. But the empty clause has no literals at all! As a result, the empty clause is always false.

The fact that the empty clause is inconsistent is the basis of the resolution method, which can be used to prove whether or not a given formula is a theorem. We’ve already run across one method of theorem-proving. That method was to check if a formula was satisfied by each and every possible truth-assignment of its propositional variables. If any truth-assignment does not satisfy the formula, we know that the formula is not a theorem. For formulas with very few variables, this method is quite effective, but when the number of variables becomes large, the number of truth-assignments that must be verified grows exponentially (recall that the number of possible truth-assignments for a formula with \( n \) propositional variables is \( 2^n \)).

The resolution method is a far different approach to theorem-proving since it doesn’t deal with explicit truth-assignments of the variables. Instead, the resolution method tries to prove that a formula is inconsistent by showing that the formula can only be true if the empty clause is true (i.e., never). In order to prove that a formula is a theorem, then, we can use the resolution method to show that its negation is inconsistent.

How does the resolution method work? Well, first, we must point out that the resolution method works with CNF formulas. Thus, we’ll assume that a formula has been converted into CNF, using our second representation (i.e., a set of clauses). With this in mind, we can now specify the resolution rule:

**Resolution Rule:** If \( C_1 \) and \( C_2 \) are clauses containing a complementary pair of literals, say \( p \in C_1 \) and \( \sim p \in C_2 \), then you can derive the new clause \((C_1 - \{p\}) \cup (C_2 - \{\sim p\})\) (i.e., remove \( p \) from \( C_1 \) and \( \sim p \) from \( C_2 \), and take the union of the two resulting sets of literals).

To understand what this means, remember that a formula in CNF is just an AND of a set of clauses. So, in order to the formula to be true, all of its clauses must be true. If we have in our set two clauses \( C_1 \) and \( C_2 \) with a complementary pair of literals (again, say \( p \in C_1 \) and \( \sim p \in C_2 \)), both \( C_1 \) and \( C_2 \) must be true for the formula to be true. We can say more than that, however. In fact, since \( p \) and \( \sim p \) can not both be true, we know that it must be the case that one of the literals in \( C_1 \), aside from \( p \), must be true or one of the literals in \( C_2 \), aside from \( \sim p \), must be true. This reasoning tells us that our resolvent \((C_1 - \{p\}) \cup (C_2 - \{\sim p\})\) must be true if the formula is true.

Consider the following examples of resolution:
(1) \( C_1 = (p), C_2 = (\sim p) \)
\((C_1 - \{p\}) \cup (C_2 - \{\sim p\}) = (), \text{ the empty clause} \)

(2) \( C_1 = (p \; q), C_2 = (\sim q \; r \; s) \)
\((C_1 - \{q\}) \cup (C_2 - \{\sim q\}) = (p \; r \; s) \)

(3) \( C_1 = (p \; q), C_2 = (\sim p \; \sim q) \)

Two resolvents can be formed from \( C_1 \) and \( C_2 \):
\((C_1 - \{p\}) \cup (C_2 - \{\sim p\}) = (q \; \sim q) \)
\((C_1 - \{q\}) \cup (C_2 - \{\sim q\}) = (p \; \sim p) \)

(In the last example, notice that we can’t remove more than a single pair of literals from the clauses – that is, we can’t remove the pair \( p \) and \( \sim p \) and the pair \( q \) and \( \sim q \) to arrive at the empty clause as a resolvent.)

With the ability to form resolvents from pairs of clauses, we’re ready to fully describe the resolution method. This method can be used to tell whether a set of clauses (i.e., a formula in CNF) is inconsistent. The method proceeds as follows:

**Resolution Method**

(1) Start with a clause set \( S \).

(2) Find all resolvents that can be made from pairs of clauses in \( S \).

(3) Add all the resulting resolvents to the set \( S \).

(4) Repeat steps (2) – (3) until either:

(a) the empty clause is produced as a resolvent
\( \Rightarrow S \) is inconsistent (unsatisfiable)

(b) no new clauses can be produced in step (2)
\( \Rightarrow S \) is consistent (satisfiable)

Whenever a resolvent clause is formed and added to the clause set, it doesn’t change the formula in any way. That is, if the formula was false for a given truth-assignment before adding the resolvent, it will still be false afterwards since one of the clauses in the original formula must have been false and still is. On the other hand, if the formula was true for a given truth-assignment before adding the resolvent, it will still be true afterwards because the truth of both clauses used to form
the resolvent results in a true resolvent (by the resolution rule). In the resolution method, then, the addition of resolvents causes us to work with several different clause sets, but each of these sets actually represents the same formula.

If we can somehow form the empty-clause resolvent, \( S \) is necessarily inconsistent since the empty clause can not be satisfied. If the empty clause can not be formed as a resolvent, it turns out that \( S \) must be consistent (the proof of this fact is beyond the scope of this discussion — see *Elements of the Theory of Computation* for more details). As long as we don’t add duplicate clauses in step (2), the resolution method will end in a finite number of steps, until the empty-clause resolvent is formed or until no new clauses can be formed.

To prove that a formula \( F \) is a theorem, the first step is to negate \( F \). Then, the resolution method can be applied to \( \sim F \). If the empty clause is formed as a resolvent, then \( \sim F \) is inconsistent; that is, \( \sim F \) is false under all possible truth-assignments. Thus, if the empty clause is formed, \( F \) is proven to be a theorem. If the empty clause is not formed, then \( F \) is consistent, meaning that \( F \) is not a theorem since there exists a truth-assignment such that \( F \) is false. Here is a complete example of theorem-proving using the resolution method:

\[
F = (A \land B) \lor (B \land C) \lor (A \land C) \lor (\sim A \land \sim B) \lor (\sim B \land \sim C) \lor (\sim A \land \sim C)
\]

In English: “Either at least two of \( A, B, C \) are true, or at least two of \( A, B, C \) are false.”

By applying DeMorgan’s Law twice, we get the negation of \( F \):

\[
\sim F = (\sim A \lor \sim B) \land (\sim B \lor \sim C) \land (\sim A \lor \sim C) \land (A \lor B) \land (B \lor C) \land (A \lor C)
\]

\( \sim F \) is already in CNF:

\[
S = \text{clause form of } \sim F = ((\sim A \land \sim B)(\sim B \land \sim C)(\sim A \land \sim C)(A \land B)(B \land C)(A \land C)(B \land C))
\]

Form the resolvent \((B \sim C)\) from \((A \land B)\) and \((\sim A \land \sim C)\):

\[
S1 = (((\sim A \land \sim B)(\sim B \land \sim C)(\sim A \land \sim C)(A \land B)(B \land C)(A \land C)(B \land C))
\]

Form the resolvent \((B)\) from \((B \sim C)\) and \((B \land C)\):

\[
S2 = ((\sim A \land \sim B)(\sim B \land \sim C)(\sim A \land \sim C)(A \land B)(B \land C)(A \land C)(B \land C))
\]

Form the resolvent \((\sim B \land C)\) from \((A \land C)\) and \((\sim A \land \sim B)\):

\[
S3 = ((\sim A \land \sim B)(\sim B \land \sim C)(\sim A \land \sim C)(A \land B)(B \land C)(A \land C)(B \land C))
\]
Form the resolvent \((\sim B)\) from \((\sim B \sim C)\) and \((\sim B \sim C)\):

\[
S4 = ( (\sim A \sim B)(\sim B \sim C)(\sim A \sim C)(A B)(B C)(A C)(B \sim C)(B)(\sim B C)(\sim B))
\]

Form the resolvent () from \((B)\) and \((\sim B)\):

Empty clause formed. \(\sim F\) is inconsistent, so \(F\) is a theorem!

An alternative way of showing a resolution-method proof is through the use of a resolution proof tree. Here is the tree corresponding to the proof above:

```
(A B)  (~A ~C)  (A C)  (~A ~B)
  (B ~C)  (B C)  (~B C)  (~B ~C)
    (B)  ()
```

Theorem Proved!

Finally, we look at a resolution proof for a formula which is not a theorem:

\[
F = ((p \lor q) \Rightarrow (p \land q))
\]

Negate \(F\):

\[
\sim F = \sim ((p \lor q) \Rightarrow (p \land q))
\]

Convert \(\sim F\) to CNF:

\[
\sim F \equiv \sim ((p \lor q) \lor (p \land q))
\]

\[
\equiv (p \lor q) \land \sim (p \land q)
\]

\[
\equiv (p \lor q) \land (\sim p \lor \sim q)
\]
5.6. PROPOSITIONAL CALCULUS PROJECTS

5.6.1 CNF Conversion (Revisited) Project

5.6.1.1 An Alternative CNF-Conversion Algorithm

The CNF-conversion algorithm proposed in the preceding section is based on both DeMorgan’s Laws and the distributive laws. An interesting fact about this algorithm is that the length of its resulting CNF formula can vary in proportion to an exponential function of the length of the original formula. That is, due to the “multiplying out” necessitated by the distributive laws, the length of the resulting CNF formula can vary proportionally to $c^n$ (where $n$ is the length of the original formula and $c$ is some positive constant). To see why this might be the case, consider the following example:

\[ F = (a \land b \land c) \lor (d \land e \land f) \lor (g \land h \land i) \]

CNF of $F$ in clause-set notation:

\[(a \land d \land g)(a \land d \land h)(a \land d \land i)(a \land e \land g)(a \land e \land h)(a \land e \land i)(a \land f \land g)(a \land f \land h)(a \land f \land i)(b \land d \land g)(b \land d \land h)(b \land d \land i)(b \land e \land g)(b \land e \land h)(b \land e \land i)(b \land f \land g)(b \land f \land h)(b \land f \land i)(c \land d \land g)(c \land d \land h)(c \land d \land i)(c \land e \land g)(c \land e \land h)(c \land e \land i)(c \land f \land g)(c \land f \land h)(c \land f \land i)\]

The original formula contains just 9 literals; the resulting CNF formula has 81!
There does exist another algorithm for CNF conversion which eliminates such exponential explosion. In this new algorithm, the length of the resulting CNF formula varies in proportion to a polynomial function of the original formula’s length. The algorithm, however, is quite different from the one described in the preceding section. In fact, the resulting CNF formula will contain literals which don’t even appear in the original formula. The algorithm builds up a CNF formula by breaking down the original formula into subformulas and describing (in logical notation) how the formula can be satisfied given its subformulas, and how these subformulas can be satisfied in terms of their own subformulas.

The best way to understand the new algorithm is to see it in action. We first represent a sample formula in prefix notation:

\[ F = (\text{implies} \ (\text{or} \ (\text{and} \ p \ (\text{not} \ q)) \ r) \ s) \]

Using a binary tree, we can diagram this formula to show its constituent subformulas. At the root (top) of the tree is the outermost operator of the formula (\text{implies}, in this case); the children of any given node are its operands:

![Binary Tree Diagram]

The portion of the tree that stems from any given node is considered a \textit{subformula}. For instance, the portion of the tree having and as its root represents the subformula \((\text{and} \ p \ (\text{not} \ q))\). (Notice that this notion of subformula implies that each of the propositional variables in the formula is a subformula and that the entire formula is itself a subformula.)

The goal of the algorithm is to describe each subformula as a simple boolean function of propositional variables (only a single such variable in the case of the \text{not} operator). By a “simple boolean function,” we mean something like \((\text{and} \ p \ q)\) or \((\text{or} \ r \ s)\), but not \((\text{and} \ (\text{not} \ p) \ q)\) or \((\text{or} \ r \ (\text{and} \ s \ t))\). To do this, we work from the bottom of the binary tree upward. When we find a node which represents an operation on propositional variables, we replace the node (and its children) with a new propositional variable (i.e., one which did not appear in the original formula). The only node fitting this description in our sample tree is the \text{not} node; we replace this node and its child by the propositional variable \(a_1\):
5.6. *PROPOSITIONAL CALCULUS PROJECTS*

We can now replace the and node and its two children by another new propositional variable, a2:

\[
\begin{array}{c}
\text{implies} \\
\text{or} \\
\text{and} \\
p \\
a1 \\
\end{array}
\]

Similarly, we can replace the or node and its children by a3:

\[
\begin{array}{c}
\text{implies} \\
\text{or} \\
a2 \\
r \\
\end{array}
\]

Finally, we replace the implies node and its children by a4, leaving us with a tree containing just the single node a4.

Having made these substitutions, we use the newly created propositional variables to rewrite our original formula. For the original formula to be true, a4 must be true. But, it is certainly not sufficient to rewrite our formula just as a4. After all, what does a4 mean? To specify what it represents, we must indicate that it is equivalent to the subformula (implies a3 s). Likewise, we must indicate that a3 is equivalent to (or a2 r), that a2 is equivalent to (and p a1), and that a1 is equivalent to (not q).

To allow for this notion of equivalence, we introduce a new boolean operator called equiv, which returns true if and only if its two operands have the same truth value. Here is the truth-table representation of equiv:

\[
\begin{array}{ccc}
\text{EQUIV} & p & q & p \equiv q \\
T & T & T \\
T & F & F \\
F & T & T \\
F & F & T \\
\end{array}
\]
(Note: We can simplify a formula turing the \textit{equiv} operator into one without it by using the logical equivalence of \((p \equiv q)\) and \(((p \rightarrow q) \land (q \rightarrow p))\).

With this new boolean operator, we rewrite our original formula using the aforementioned equivalencies:

\[
F = (\text{and } a4 \\
(\text{equiv } a4 (\text{implies } a3 s)) \\
(\text{equiv } a3 (\text{or } a2 r)) \\
(\text{equiv } a2 (\text{and } p a1)) \\
(\text{equiv } a1 (\text{not } q)))
\]

It still seems as if we haven’t progressed very far in our effort to transform the original formula into CNF, but we’re actually just one simple step away. Notice that \(F\) is an \textit{and} of several subformulas. If we can transform each of these subformulas to CNF, we’ll be done. This is where our algorithm proves to be quite useful; notice that there are only four types of subformulas which can be created by our algorithm:

\begin{enumerate}
  \item (\text{equiv } p (\text{not } q))
  \item (\text{equiv } p (\text{or } q r))
  \item (\text{equiv } p (\text{and } q r))
  \item (\text{equiv } p (\text{implies } q r))
\end{enumerate}

(where \(p, q, r\) are arbitrary propositional variables)

All that is necessary to convert the original formula to CNF, then, is to have CNF conversions for each of these four possibilities. These conversions are as follows (you may verify them if you wish):

\begin{center}
\begin{tabular}{ll}
\text{FORMULA} & \text{CNF (in clause-set notation)} \\
\hline
(\text{equiv } p (\text{not } q)) & (((p q) ((not p) (not q)))) \\
(\text{equiv } p (\text{or } q r)) & (((not p) q r) (p (not q)) (p (not r))) \\
(\text{equiv } p (\text{and } q r)) & ((p (not q) (not r)) ((not p) q) ((not p) r)) \\
(\text{equiv } p (\text{implies } q r)) & (((not p) (not q) r) (p q) (p (not r))) \\
\end{tabular}
\end{center}

These four conversions allow us to represent the CNF of \(F\) using clause-set notation:

\[
((a4) \\
((not a4) (not a3) s) a4 a3 a4 (not s)) \\
((not a3) a2 r) (a3 (not a2)) (a3 (not r)) \\
(a2 (not p) (not a1)) ((not a2) p) ((not a2) a1) \\
(a1 q) ((not a1) (not q)))
\]
5.6. PROPOSITIONAL CALCULUS PROJECTS

(Note: Recall that the function normalize-clause-set can be used to normalize this CNF representation.)

5.6.1.2 Implementation of the Algorithm in LISP

For this project, you will write a function make-cnf-new which can serve as a replacement for the existing make-cnf function. make-cnf-new should use the algorithm described above in order to convert a formula in prefix notation to a clause-set which represents its CNF. If you think recursively, the implementation of make-cnf-new should be relatively straightforward. The function should take a tree (subformula) as its argument; it can access left and right children (if there are any) by simply looking at the cadr and caddr of the tree representation.

Since our algorithm works only on binary trees, an additional conversion of form must take place before the algorithm is invoked. In particular, we have allowed and and or to have more than two operands. You should write a function change-to-binary which takes a formula in prefix notation and returns an equivalent formula (also in prefix notation) which contains only binary operations. For instance, the formula

\[(\text{and } p (\text{or } q r s) t)\]

would be changed to

\[(\text{and } (\text{and } p (\text{or } (\text{or } q r) s) t).)\]

(Note: You may assume that and and or have at least two operands. That is, don’t worry about cases in which either of these operations takes only a single operand or no operand at all.)

Once converted by change-to-binary, a formula can be thought of as a binary tree and, thus, can be converted to CNF by our algorithm. The one complication involved in the algorithm is that new propositional variables have to be created on the fly. That is, in LISP, you will have to create a new symbol each time that you want to replace a subformula by a new propositional variable. The built-in function gensym, called with no arguments, should do the trick (i.e., it will create a symbol which has not yet been used in the environment). However, since gensym creates a new symbol each and every time it is called, you will want to save the symbol created so that you can use it appropriately. To do this, you might have code like

\[
\text{(setq tempvar (gensym))}
\]

This setq serves to store the new symbol as tempvar’s value; from there, you can retrieve the new symbol with the expression \(\text{(eval tempvar)}\).

Feel free to use any of the code from the preceding section in order to write the code for this project. With make-cnf-new written, you can write a another function make-clause-set-new (to normalize the representation) in order to compare the performance of the two CNF-conversion algorithms. (Note: Try out the new algorithm on the example we gave at the beginning of this project. The number of literals in the new algorithm’s result should be 56.)
5.6.2 Resolution-Method Theorem Prover

5.6.2.1 The Basic Theorem Prover

For this project, you will write the code necessary to implement a resolution-method theorem prover in LISP. The main function, `prove`, should take an arbitrary formula in prefix notation as its only argument. Your theorem prover should use the code from the preceding section which already does the work of transforming formulas into conjunctive normal form and normalizing clauses.

The resolution procedure is actually nothing more than a state-space search problem, with the goal state being a clause set which contains the empty clause. In the state-space we are searching, a transition from one set of clauses to another is done by choosing any two resolvable clauses and adding their resolvent. There are many strategies for picking any two clauses to resolve. The suggested strategy for the theorem prover is breadth-first search since it is guaranteed to find a solution (i.e., “Formula is a theorem” or “Formula is not a theorem”). In breadth-first search, you will form all possible resolvents from a set of clauses before adding any of them into the set and repeating (if necessary).

Here is a brief summary of how the theorem prover should work:

1. At the beginning, the program should print the initial clauses in the CNF of the formula’s negation.

2. Whenever a new resolvent is produced, the program should print the two clauses involved and their resolvent, in a neat and readable manner.

3. If the empty resolvent (represented by `nil` in LISP) is found, the program should exit (as soon as possible) and report that the theorem has been proven.

4. When no more resolvents can be found, the program should exit and report that the formula is not a theorem.

5. The return value from a successful proof should be non--`nil`; the return value from an unsuccessful one should be `nil`. (In this way, your program could be used as a predicate in some larger program.)

5.6.2.2 Optimizing the Theorem Prover

Since the goal in a search problem is to find the solution as quick as possible, it’s useful to optimize the search by applying knowledge of the problem at hand — in this case, a problem involving propositional calculus. We suggest below two refinements which should be made to the basic algorithm to (hopefully!) make the strategy more efficient:
5.6. PROPOSITIONAL CALCULUS PROJECTS

(1) Don’t attempt to resolve clauses that are obviously unresolvable. By “obviously unresolvable,” we mean clauses that we’ve tried to resolve already. For instance, if we start out with a set of three clauses, we first try to form all resolvents possible from these clauses; however, in the next iteration of the algorithm, we don’t want to check again for resolvents that can be formed from a pair of clauses contained in the original set of three clauses. The easiest way to avoid such repetition is to keep track of three separate clause sets at all times—old-clause-set, current-clause-set, and new-clause-set. The set old-clause-set should contain all those clauses which have already been checked against each other for possible resolvents; current-clause-set should contain those clauses which have not yet been checked against each other or against the clauses in old-clause-set; and, new-clause-set should contain those clauses which are newly formed. In terms of the search, current-clause-set should be initialized to the clauses in the CNF of the formula’s negation, while both old-clause-set and new-clause-set should be initialized to nil. Then, at each iteration of the search:

- new-clause-set becomes the set of clauses resulting from (1) resolving the clauses in current-clause-set with the clauses in current-clause-set and (2) resolving the clauses in current-clause-set with the clauses in old-clause-set
- old-clause-set becomes the union of old-clause-set and current-clause-set
- current-clause-set becomes the newly formed new-clause-set

(2) Don’t form a tautology as a resolvent. A tautology is any clause that contains both a literal and its negation—having both makes the clause true under any truth-assignment. Since we are trying to show a set of clauses to be inconsistent, tautologies can’t help. To look at it mechanically, if we resolve a tautology such as \((p \sim p \ldots)\) with some other clause \(C\), we get either another tautology (if the resolution doesn’t remove \(p\) from one of the clauses) or a clause containing all of \(C\)’s literals (if the resolution does remove \(p\) from one of the clauses).

5.6.2.3 Optional Additions

- Our implementation of propositional formulas allows for only four boolean operators (and, or, not, implies). Modify the implementation so that it will also support the operators equiv (“equivalent”) and xor (“exclusive-or”). Both operators are binary (i.e., they accept just two operands). equiv returns true if and only if its two operands have the same truth values; xor returns true if and only if exactly one of its operands is true. They can be expressed in terms of the other operators as follows:
(equiv p q) \equiv (\text{and} (\text{implies} p q) (\text{implies} q p))
(xor p q) \equiv (\text{or} (\text{and} p (\text{not} q)) (\text{and} q (\text{not} p)))

- When using the resolution method, the goal is to find the empty clause. Intuition suggests that if there is a choice between dealing with short clauses and dealing with long clauses, we’re better off dealing with short ones. (After all, there’s no hope of getting to the empty clause if we don’t work with some short clauses sooner or later.) In this regard, unit clauses (clauses with a single literal) are especially attractive since only a unit clause can produce a child (resolvent) shorter than the clause with which it’s being resolved. Try to optimize your theorem prover by treating short clauses (especially unit clauses) preferentially.

- Get rid of any subsumed clauses. A subsumed clause is a clause such that another clause already in the set of clauses is its subset; for instance, if both \((p \land q)\) and \((p \land q \land r)\) are in the set of clauses, then \((p \land q \land r)\) is a subsumed clause. A subsumed clause can be of no more help in the resolution process than the clause which subsumes it (why?).

- When the formula to be proved is an implication \((\alpha \Rightarrow \beta)\) and \(\alpha\) is a conjunction of “axioms” which are known not to be inconsistent, the only way to get an inconsistency is to involve a clause that arises from \(\beta\). (As long as we only deal with clauses arising from \(\alpha\), we are just generating consequences of the axioms.) In this special case, then, perhaps one clause in every resolution attempted should trace its “ancestry” back to \(\beta\). For this optimization, you would presumably want to change the calling convention to something like (prove-impl \(\alpha \beta\)).

- The optimizations discussed until now have involved reducing the number of resolvents to be produced, but the program can also be made much more efficient in other ways — for example, by making destructive changes to list structure instead of consing up new structure, by sorting less often, etc. Try some of these low-level optimizations and see what type of impact they have on running time.
Chapter 6

Data Representation

by Daniel J. Ellard

In order to understand how a computer is able to manipulate data and perform computations, you must first understand how data is represented by a computer.

At the lowest level, the indivisible unit of data in a computer is a bit. A bit represents a single binary value, which may be either 1 or 0. In different contexts, a bit value of 1 and 0 may also be referred to as “true” and “false”, “yes” and “no”, “high” and “low”, “set” and “not set”, or “on” and “off”.

The decision to use binary values, rather than something larger (such as decimal values) was not purely arbitrary– it is due in a large part to the relative simplicity of building electronic devices that can manipulate binary values.

6.1 Representing Integers

6.1.1 Unsigned Binary Numbers

While the idea of a number system with only two values may seem odd, it is actually very similar to the decimal system we are all familiar with, except that each digit is a bit containing a 0 or 1 rather than a number from 0 to 9. (The word “bit” itself is a contraction of the words “binary digit”)

For example, figure 6.1 shows several binary numbers, and the equivalent decimal numbers.

In general, the binary representation of $2^k$ has a 1 in binary digit $k$ (counting from the right, starting at 0) and a 0 in every other digit. (For notational convenience, the $i$th bit of a binary number $A$ will be denoted as $A_i$.)

The binary representation of a number that is not a power of 2 has the bits set corresponding to
the powers of two that sum to the number: for example, the decimal number 6 can be expressed in terms of powers of 2 as \(1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0\), so it is written in binary as 110.

An eight-digit binary number is commonly called a *byte*. In this text, binary numbers will usually be written as bytes (i.e. as strings of eight binary digits). For example, the binary number 101 would usually be written as 00000101 – a 101 padded on the left with five zeros, for a total of eight digits.

Whenever there is any possibility of ambiguity between decimal and binary notation, the *base* of the number system (which is 2 for binary, and 10 for decimal) is appended to the number as a subscript. Therefore, 101\textsubscript{2} will always be interpreted as the binary representation for five, and never the decimal representation of one hundred and one (which would be written as 101\textsubscript{10}).

### 6.1.1.1 Conversion of Binary to Decimal

To convert an unsigned binary number to a decimal number, add up the decimal values of the powers of 2 corresponding to bits which are set to 1 in the binary number. Algorithm 6.1 shows a method to do this. Some examples of conversions from binary to decimal are given in figure 6.2.

Since there are \(2^n\) unique sequences of \(n\) bits, if all the possible bit sequences of length \(n\) are used, starting from zero, the largest number will be \(2^n - 1\).

### 6.1.1.2 Conversion of Decimal to Binary

An algorithm for converting a decimal number to binary notation is given in algorithm 6.2.
Algorithm 6.1 To convert a binary number to decimal.

- Let $X$ be a binary number, $n$ digits in length, composed of bits $X_{n-1} \cdots X_0$.
- Let $D$ be a decimal number.
- Let $i$ be a counter.

1. Let $D = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
   - If $X_i == 1$ (i.e. if bit $i$ in $X$ is 1), then set $D = (D + 2^i)$.
   - Set $i = (i + 1)$.

---

Figure 6.2: Examples of Conversion from Binary to Decimal

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>$2^0$</td>
</tr>
<tr>
<td>00000101</td>
<td>$2^2 + 2^0$</td>
</tr>
<tr>
<td>00001100</td>
<td>$2^2 + 2^1$</td>
</tr>
<tr>
<td>00101101</td>
<td>$2^5 + 2^3 + 2^2 + 2^0$</td>
</tr>
<tr>
<td>10110000</td>
<td>$2^7 + 2^5 + 2^4$</td>
</tr>
</tbody>
</table>
Algorithm 6.2 To convert a positive decimal number to binary.

- Let \( X \) be an unsigned binary number, \( n \) digits in length.
- Let \( D \) be a positive decimal number, no larger than \( 2^n - 1 \).
- Let \( i \) be a counter.

1. Let \( X = 0 \) (set all bits in \( X \) to 0).
2. Let \( i = (n - 1) \).
3. While \( i \geq 0 \) do:
   a. If \( D \geq 2^i \), then
      - Set \( X_i = 1 \) (i.e. set bit \( i \) of \( X \) to 1).
      - Set \( D = (D - 2^i) \).
   b. Set \( i = (i - 1) \).

6.1.1.3 Addition of Unsigned Binary Numbers

Addition of binary numbers can be done in exactly the same way as addition of decimal numbers, except that all of the operations are done in binary (base 2) rather than decimal (base 10). Algorithm 6.3 gives a method which can be used to perform binary addition.

When algorithm 6.3 terminates, if \( c \) is not 0, then an overflow has occurred– the resulting number is simply too large to be represented by an \( n \)-bit unsigned binary number.

6.1.2 Signed Binary Numbers

The major drawback with the representation that we’ve used for unsigned binary numbers is that it doesn’t include a way to represent negative numbers.

There are a number of ways to extend the unsigned representation to include negative numbers. One of the easiest is to add an additional bit to each number that is used to represent the sign of the number– if this bit is 1, then the number is negative, otherwise the number is positive (or vice versa). This is analogous to the way that we write negative numbers in decimal– if the first symbol in the number is a negative sign, then the number is negative, otherwise the number is positive.

Unfortunately, when we try to adapt the algorithm for addition to work properly with this
Algorithm 6.3 Addition of binary numbers (unsigned).

- Let $A$ and $B$ be a pair of $n$-bit binary numbers.
- Let $X$ be a binary number which will hold the sum of $A$ and $B$.
- Let $c$ and $\hat{c}$ be carry bits.
- Let $i$ be a counter.
- Let $s$ be an integer.

1. Let $c = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
   (a) Set $s = A_i + B_i + c$.
   (b) Set $X_i$ and $\hat{c}$ according to the following rules:
       - If $s == 0$, then $X_i = 0$ and $\hat{c} = 0$.
       - If $s == 1$, then $X_i = 1$ and $\hat{c} = 0$.
       - If $s == 2$, then $X_i = 0$ and $\hat{c} = 1$.
       - If $s == 3$, then $X_i = 1$ and $\hat{c} = 1$.
   (c) Set $c = \hat{c}$.
   (d) Set $i = (i + 1)$. 
representation, this apparently simple method turns out to cause some trouble. Instead of simply adding the numbers together as we do with unsigned numbers, we now need to consider whether the numbers being added are positive or negative. If one number is positive and the other negative, then we actually need to do subtraction instead of addition, so we’ll need to find an algorithm for subtraction. Furthermore, once we’ve done the subtraction, we need to compare the the unsigned magnitudes of the numbers to determine whether the result is positive or negative.

Luckily, there is a representation that allows us to represent negative numbers in such a way that addition (or subtraction) can be done easily, using algorithms very similar to the ones that we already have. The representation that we will use is called two’s complement notation.

To introduce two’s complement, we’ll start by defining, in algorithm 6.4, the algorithm that is used to compute the negation of a two’s complement number.

**Algorithm 6.4** Negation of a two’s complement number.

1. Let $\bar{x} = \text{the logical complement of } x$.

   The logical complement (also called the one’s complement) is formed by flipping all the bits in the number, changing all of the 1 bits to 0, and vice versa.

2. Let $X = \bar{x} + 1$.

   If this addition *overflows*, then the overflow bit is discarded.

By the definition of two’s complement, $X \equiv -x$.

Figure 6.3 shows the process of negating several numbers. Note that the negation of zero is zero.

This representation has several useful properties:

- The leftmost (most significant) bit also serves as a sign bit; if 1, then the number is negative, if 0, then the number is positive or zero.

- The rightmost (least significant) bit of a number always determines whether or not the number is odd or even– if bit 0 is 0, then the number is even, otherwise the number is odd.

- The largest positive number that can be represented in two’s complement notation in an $n$-bit binary number is $2^{n-1} - 1$. For example, if $n = 8$, then the largest positive number is $01111111 = 2^7 - 1 = 127$. 

6.1. REPRESENTING INTEGERS

Figure 6.3: Examples of Negation Using Two’s Complement

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00000110</td>
<td>= 6</td>
<td></td>
</tr>
<tr>
<td>1’s complement</td>
<td>11111001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 1</td>
<td>11111010</td>
<td>= -6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11111010</td>
<td>= -6</td>
<td></td>
</tr>
<tr>
<td>1’s complement</td>
<td>00000101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 1</td>
<td>00000110</td>
<td>= 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00000000</td>
<td>= 0</td>
<td></td>
</tr>
<tr>
<td>1’s complement</td>
<td>11111111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 1</td>
<td>00000000</td>
<td>= 0</td>
<td></td>
</tr>
</tbody>
</table>

- Similarly, the “most negative” number is \(-2^{n-1}\), so if \(n = 8\), then it is 10000000, which is \(-2^7 = -128\). Note that the negative of the most negative number (in this case, 128) cannot be represented in this notation.

### 6.1.2.1 Addition and Subtraction of Signed Binary Numbers

The same addition algorithm that was used for unsigned binary numbers also works properly for two’s complement numbers.

\[
\begin{align*}
00000101 + 11110101 &= 11111010 \\
00000101 + 11101001 &= 11110110 \\
00000101 + 11111111 &= 10010001
\end{align*}
\]

Subtraction is also done in a similar way: to subtract A from B, take the two’s complement of A and then add this number to B.

The conditions for detecting overflow are different for signed and unsigned numbers, however. If we use algorithm 6.3 to add two unsigned numbers, then if \(c\) is 1 when the addition terminates, this indicates that the result has an absolute value too large to fit the number of bits allowed. With signed numbers, however, \(c\) is not relevant, and an overflow occurs when the signs of both numbers being added are the same but the sign of the result is opposite. If the two numbers being added have opposite signs, however, then an overflow cannot occur.
For example, consider the sum of 1 and −1:

\[
\begin{array}{c}
00000001 = 1 \\
+ \quad 11111111 = -1 \\
\hline
00000000 = 0 \text{ Correct!}
\end{array}
\]

In this case, the addition will overflow, but it is not an error, since the result that we get (without considering the overflow) is exactly correct.

On the other hand, if we compute the sum of 127 and 1, then a serious error occurs:

\[
\begin{array}{c}
01111111 = 127 \\
+ \quad 00000001 = 1 \\
\hline
10000000 = -128 \text{ Uh-oh!}
\end{array}
\]

Therefore, we must be very careful when doing signed binary arithmetic that we take steps to detect bogus results. In general:

- If \( A \) and \( B \) are of the same sign, but \( A + B \) is of the opposite sign, then an overflow or wraparound error has occurred.
- If \( A \) and \( B \) are of different signs, then \( A + B \) will never overflow or wraparound.

### 6.1.2.2 Shifting Signed Binary Numbers

Another useful property of the two’s complement notation is the ease with which numbers can be multiplied or divided by two. To multiply a number by two, simply shift the number “up” (to the left) by one bit, placing a 0 in the least significant bit. To divide a number in half, simply shift the the number “down” (to the right) by one bit (but do not change the sign bit).

Note that in the case of odd numbers, the effect of shifting to the right one bit is like dividing in half, rounded towards \(-\infty\), so that 51 shifted to the right one bit becomes 25, while -51 shifted to the right one bit becomes -26.
6.1. REPRESENTING INTEGERS

Figure 6.4: Hexadecimal and Octal

<table>
<thead>
<tr>
<th>Binary</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Hex</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Octal</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Hex</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>Octal</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

00000001 = 1  
Double 00000010 = 2  
Halve 00000000 = 0  

00110011 = 51  
Double 01100110 = 102  
Halve 00011001 = 25  

11001101 = -51  
Double 10011010 = -102  
Halve 11100110 = -26

6.1.2.3 Hexadecimal Notation

Writing numbers in binary notation can soon get tedious, since even relatively small numbers require many binary digits to express. A more compact notation, called hexadecimal (base 16), is usually used to express large binary numbers. In hexadecimal, each digit represents four unsigned binary digits.

Another notation, which is not as common currently, is called octal and uses base eight to represent groups of three bits. Figure 6.4 show examples of binary, decimal, octal, and hexadecimal numbers.

For example, the number 200_{10} can be written as 11001000_{2}, C8_{16}, or 310_{8}. 
CHAPTER 6. DATA REPRESENTATION

6.2 Representing Characters

Just as sequences of bits can be used to represent numbers, they can also be used to represent the letters of the alphabet, as well as other characters.

Since all sequences of bits represent numbers, one way to think about representing characters by sequences of bits is to choose a number that corresponds to each character. The most popular correspondence currently is the ASCII character set. ASCII, which stands for the American Standard Code for Information Interchange, uses 7-bit integers to represent characters, using the correspondence shown in table 6.5.

Figure 6.5: The ASCII Character Set

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>NUL</td>
<td>01</td>
<td>SOH</td>
<td>02</td>
<td>STX</td>
<td>03</td>
<td>ETX</td>
<td>04</td>
<td>EOT</td>
<td>05</td>
<td>ENQ</td>
<td>06</td>
<td>ACK</td>
<td>07</td>
<td>BEL</td>
</tr>
<tr>
<td>08</td>
<td>BS</td>
<td>09</td>
<td>HT</td>
<td>0A</td>
<td>NL</td>
<td>0B</td>
<td>VT</td>
<td>0C</td>
<td>NP</td>
<td>0D</td>
<td>CR</td>
<td>0E</td>
<td>SO</td>
<td>0F</td>
<td>SI</td>
</tr>
<tr>
<td>10</td>
<td>DLE</td>
<td>11</td>
<td>DC1</td>
<td>12</td>
<td>DC2</td>
<td>13</td>
<td>DC3</td>
<td>14</td>
<td>DC4</td>
<td>15</td>
<td>NAK</td>
<td>16</td>
<td>SYN</td>
<td>17</td>
<td>ETB</td>
</tr>
<tr>
<td>18</td>
<td>CAN</td>
<td>19</td>
<td>EM</td>
<td>1A</td>
<td>SUB</td>
<td>1B</td>
<td>ESC</td>
<td>1C</td>
<td>FS</td>
<td>1D</td>
<td>GS</td>
<td>1E</td>
<td>RS</td>
<td>1F</td>
<td>US</td>
</tr>
<tr>
<td>20</td>
<td>SP</td>
<td>21</td>
<td>!</td>
<td>22</td>
<td>&quot;</td>
<td>23</td>
<td>#</td>
<td>24</td>
<td>$</td>
<td>25</td>
<td>%</td>
<td>26</td>
<td>&amp;</td>
<td>27</td>
<td>’</td>
</tr>
<tr>
<td>28</td>
<td>(</td>
<td>29</td>
<td>)</td>
<td>2A</td>
<td>*</td>
<td>2B</td>
<td>+</td>
<td>2C</td>
<td>,</td>
<td>2D</td>
<td>-</td>
<td>2E</td>
<td>.</td>
<td>2F</td>
<td>/</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>31</td>
<td>1</td>
<td>32</td>
<td>2</td>
<td>33</td>
<td>3</td>
<td>34</td>
<td>4</td>
<td>35</td>
<td>5</td>
<td>36</td>
<td>6</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
<td>39</td>
<td>9</td>
<td>3A</td>
<td>:</td>
<td>3B</td>
<td>;</td>
<td>3C</td>
<td>&lt;</td>
<td>3D</td>
<td>=</td>
<td>3E</td>
<td>&gt;</td>
<td>3F</td>
<td>?</td>
</tr>
<tr>
<td>40</td>
<td>@</td>
<td>41</td>
<td>A</td>
<td>42</td>
<td>B</td>
<td>43</td>
<td>C</td>
<td>44</td>
<td>D</td>
<td>45</td>
<td>E</td>
<td>46</td>
<td>F</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>48</td>
<td>H</td>
<td>49</td>
<td>I</td>
<td>4A</td>
<td>J</td>
<td>4B</td>
<td>K</td>
<td>4C</td>
<td>L</td>
<td>4D</td>
<td>M</td>
<td>4E</td>
<td>N</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>50</td>
<td>P</td>
<td>51</td>
<td>Q</td>
<td>52</td>
<td>R</td>
<td>53</td>
<td>S</td>
<td>54</td>
<td>T</td>
<td>55</td>
<td>U</td>
<td>56</td>
<td>V</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>58</td>
<td>X</td>
<td>59</td>
<td>Y</td>
<td>5A</td>
<td>Z</td>
<td>5B</td>
<td>[</td>
<td>5C</td>
<td>]</td>
<td>5D</td>
<td>^</td>
<td>5E</td>
<td>_</td>
<td>5F</td>
<td>`</td>
</tr>
<tr>
<td>60</td>
<td>'</td>
<td>61</td>
<td>a</td>
<td>62</td>
<td>b</td>
<td>63</td>
<td>c</td>
<td>64</td>
<td>d</td>
<td>65</td>
<td>e</td>
<td>66</td>
<td>f</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>68</td>
<td>h</td>
<td>69</td>
<td>i</td>
<td>6A</td>
<td>j</td>
<td>6B</td>
<td>k</td>
<td>6C</td>
<td>l</td>
<td>6D</td>
<td>m</td>
<td>6E</td>
<td>n</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>70</td>
<td>p</td>
<td>71</td>
<td>q</td>
<td>72</td>
<td>r</td>
<td>73</td>
<td>s</td>
<td>74</td>
<td>t</td>
<td>75</td>
<td>u</td>
<td>76</td>
<td>v</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>78</td>
<td>x</td>
<td>79</td>
<td>y</td>
<td>7A</td>
<td>z</td>
<td>7B</td>
<td>{</td>
<td>7C</td>
<td></td>
<td></td>
<td>7D</td>
<td>}</td>
<td>7E</td>
<td>-</td>
<td>7F</td>
</tr>
</tbody>
</table>

When the ASCII character set was chosen, some care was taken to organize the way that characters are represented in order to make them easy for a computer to manipulate. For example, all of the letters of the alphabet are arranged in order, so that sorting characters into alphabetical order is the same as sorting in numerical order. In addition, different classes of characters are arranged to have useful relations. For example, to convert the code for a lowercase letter to the code for the same letter in uppercase, simply set the 6th bit of the code to 0 (or subtract 32). ASCII is by no means the only character set to have similar useful properties, but it has emerged as the standard.
6.3. REPRESENTING RATIONAL NUMBERS

The ASCII character set does have some important limitations, however. One problem is that the character set only defines the representations of the characters used in written English. This causes problems with using ASCII to represent other written languages. In particular, there simply aren’t enough bits to represent all the written characters of languages with a larger number of characters (such as Chinese or Japanese). Already new character sets which address these problems (and can be used to represent characters of many languages side by side) are being proposed, and eventually there will unquestionably be a shift away from ASCII to a new multilanguage standard.1

6.3 Representing Rational Numbers

Up to this point, we’ve only addressed the problem of representing integers. Many important problems, however, require real numbers to solve. Let’s think about how these numbers can be represented using binary digits.

The first observation that we might make is that a rational number can always be represented as the sum of an integral number and a rational number between 0 and 1. We already know how we can represent the integral part, but we need to make a choice about how to represent the “leftover” piece which will be between 0 and 1.

Let \( R \) be the leftover fraction, and let \( n \) be the number of bits that we have to represent it. One representation would be to represent the fraction by the integer \( X \), where \( X \) is the unsigned multiple of \( 2^{-n} \) which is closest to \( R \). Once we have \( X \), we append this to the binary number which represents the integral portion of the original number.

For example, if we let \( n \) be 8, then:

<table>
<thead>
<tr>
<th>Number</th>
<th>Integer ( I )</th>
<th>Fraction ( R )</th>
<th>( X )</th>
<th>( I.X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>2</td>
<td>1/2</td>
<td>128</td>
<td>000000010.10000000</td>
</tr>
<tr>
<td>0.75</td>
<td>0</td>
<td>3/4</td>
<td>196</td>
<td>00000000.11000000</td>
</tr>
<tr>
<td>10.01</td>
<td>10</td>
<td>1/100</td>
<td>3</td>
<td>00001010.00000011</td>
</tr>
</tbody>
</table>

This representation isn’t too bad, and allows us to use the same algorithms that we used for all the other binary numbers—just ignore the “point” in the middle of the numbers.

Unfortunately, this representation has a major problem— it isn’t good for very large or very small numbers. In a very large number, we’ll need a lot of digits to the left of the decimal place, while for very small numbers, we’ll need a lot more to the right. As long as we’re concerned about the number of bits that our representation uses (i.e. for the foreseeable future), this representation

---

1This shift will break many, many existing programs. Converting all of these programs will keep many, many programmers busy for some time.
has serious problems. What we really want is some way to represent a number that allows the decimal “point” (or whatever is used to represent the decimal point) to be wherever is most appropriate. A representation that has this property is called a floating point representation.

In the floating point representations used most commonly today, a number is represented by three sequences of bits:

1. The sign bit, which specifies whether the number is positive or negative.
2. The mantissa, which is an unsigned number from 0 to \(2^m - 1\), where \(m\) is the number of bits allotted to the mantissa.
   In some texts, the mantissa is also referred to as the significand.
3. The exponent, which is a signed binary number, which is the power of 2 to multiply the mantissa by.

The most common representation for floating point numbers today is the IEEE 754 floating-point standard. This standard defines a 32-bit floating point number \(R\) to be represented in the following way:

- The high-order bit represents \(S\), the sign of the number. A 1 indicates that the number is negative, 0 positive.
- The next 8 bits represent \(E\), the exponent, which is stored as an 8-bit unsigned number.
- The low-order 23 bits represent \(M\), the mantissa, which is stored as a 23-bit unsigned number.

The exponent \(E\) represents the power of two by which it is necessary to divide \(R\) in order to get a number in the range from 1 to 2, and \(M\) is the fractional portion of this number. \(E\) is represented by an unsigned number, but in order to allow it to represent negative numbers it is biased by subtracting 127. Therefore, the value represented by a number in this format is:

\[
R = (-1)^S \times (1 + M) \times 2^{E-127}
\]

### 6.4 Representing Programs

Just as groups of bits can be used to represent numbers, they can also be used to represent instructions for a computer to perform. Unlike the two’s complement notation for integers, which is a standard representation used by nearly all computers, the representation of instructions, and even the set of instructions, varies widely from one type of computer to another.
6.5. MEMORY ORGANIZATION

The MIPS architecture, which is the focus of later chapters in this document, uses a relatively simple and straightforward representation. Each instruction is exactly 32 bits in length, and consists of several bit fields, as depicted in figure 6.6.

Figure 6.6: MIPS R2000 Instruction Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>op</td>
<td>reg1</td>
<td>reg2</td>
<td>des</td>
<td>shift</td>
<td>funct</td>
</tr>
<tr>
<td>Immediate</td>
<td>op</td>
<td>reg1</td>
<td>reg2</td>
<td></td>
<td></td>
<td>16-bit constant</td>
</tr>
<tr>
<td>Jump</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26-bit constant</td>
</tr>
</tbody>
</table>

The first six bits (reading from the left, or high-order bits) of each instruction are called the op field. The op field determines whether the instruction is a register, immediate, or jump instruction, and how the rest of the instruction should be interpreted. Depending on what the op is, parts of the rest of the instruction may represent the names of registers, constant memory addresses, 16-bit integers, or other additional qualifiers for the op.

If the op field is 0, then the instruction is a register instruction, which generally perform an arithmetic or logical operations. The funct field specifies the operation to perform, while the reg1 and reg2 represent the registers to use as operands, and the des field represents the register in which to store the result. For example, the 32-bit hexadecimal number 0x02918020 represents, in the MIPS instruction set, the operation of adding the contents of registers 20 and 17 and placing the result in register 16.

<table>
<thead>
<tr>
<th>Field</th>
<th>Width</th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>0</td>
<td>20</td>
<td>17</td>
<td>16</td>
<td>0</td>
<td>add</td>
</tr>
<tr>
<td>Binary</td>
<td>000000</td>
<td>10100</td>
<td>10001</td>
<td>10000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

If the op field is not 0, then the instruction may be either an immediate or jump instruction, depending on the value of the op field.

6.5 Memory Organization

We’ve seen how sequences of binary digits can be used to represent numbers, characters, and instructions. In a computer, these binary digits are organized and manipulated in discrete groups, and these groups are said to be the memory of the computer.
6.5.1 Units of Memory

The smallest of these groups, on most computers, is called a byte. On nearly all currently popular computers a byte is composed of 8 bits.

The next largest unit of memory is usually composed of 16 bits. What this unit is called varies from computer to computer– on smaller machines, this is often called a word, while on newer architectures that can handle larger chunks of data, this is called a halfword.

The next largest unit of memory is usually composed of 32 bits. Once again, the name of this unit varies– on smaller machines, it is referred to as a long, while on newer and larger machines it is called a word.

Finally, on the newest machines, the computer also can handle data in groups of 64 bits. On a smaller machine, this is known as a quadword, while on a larger machine this is known as a long.

6.5.1.1 Historical Perspective

There have been architectures that have used nearly every imaginable word size– from 6-bit bytes to 9-bit bytes, and word sizes ranging from 12 bits to 48 bits. There are even a few architectures that have no fixed word size at all (such as the CM-2) or word sizes that can be specified by the operating system at runtime.

Over the years, however, most architectures have converged on 8-bit bytes and 32-bit long-words. An 8-bit byte is a good match for the ASCII character set (which has some popular extensions that require 8 bits), and a 32-bit word has been, at least until recently, large enough for most practical purposes.

6.5.2 Addresses and Pointers

Each unique byte\(^2\) of the computer’s memory is given a unique identifier, known as its address. The address of a piece of memory is often referred to as a pointer to that piece of memory– the two terms are synonymous, although there are many contexts where one is commonly used and the other is not.

The memory of the computer itself can often be thought of as a large array (or group of arrays) of bytes of memory. In this model, the address of each byte of memory is simply the index of the memory array location where that byte is stored.

\(^2\)In some computers, the smallest distinct unit of memory is not a byte. For the sake of simplicity, however, this section assumes that the smallest distinct unit of memory on the computer in question is a byte.
6.6 Exercises

6.6.1

Complete the following table:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>123</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>01101100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Octal</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex</td>
<td>3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASCII</td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.6.2

1. Invent an algorithm for multiplying two unsigned binary numbers. You may find it easiest to start by thinking about multiplication of decimal numbers (there are other ways as well, but you should start on familiar ground).

6.6.3

1. Invent an algorithm for dividing two unsigned binary numbers. You may find it easiest to start by thinking about long division of decimal numbers.

2. Your TF complains that the division algorithm you invented to solve the previous part of this problem is too slow. She would prefer an algorithm that gets an answer that is “reasonably close” to the right answer, but which may take considerably less time to compute. Invent an algorithm that has this property. Find the relationship between “reasonably close” and the speed of your algorithm.
Chapter 7

A MIPS Tutorial

by Daniel J. Ellard

This section is a quick tutorial for MIPS assembly language programming and the SPIM environment. For more detailed information about the MIPS instruction set and the SPIM environment, consult chapter 8 of this book, and SPIM S20: A MIPS R2000 Simulator by James Larus.

Other references include Computer Organization and Design, by David Patterson and John Hennessy (which includes an expanded version of James Larus’ SPIM documentation as appendix A), and MIPS R2000 RISC Architecture by Gerry Kane.

7.1 What is Assembly Language?

As we saw in the previous chapter, computer instructions can be represented as sequences of bits. Generally, this is the lowest possible level of representation for a program—each instruction is equivalent to a single, indivisible action of the CPU. This representation is called machine language, since it is the only form that can be “understood” directly by the machine.

A slightly higher-level representation (and one that is much easier for humans to use) is called assembly language. Assembly language is very closely related to machine language, and there is usually a straightforward way to translate programs written in assembly language into machine language. (This algorithm is usually implemented by a program called the assembler.) Because of the close relationship between machine and assembly languages, each different machine architecture usually has its own assembly language (in fact, each architecture may have several), and each is unique.

1For many years, considerable effort was spent trying to develop a portable assembly which could generate machine language for a wide variety of architectures. Eventually, these efforts were abandoned as hopeless.
The advantage of programming in assembler (rather than machine language) is that assembly language is much easier for a human to read and understand. For example, the MIPS machine language instruction for adding the contents of registers 20 and 17 and placing the result in register 16 is the integer 0x02918020. This representation is fairly impenetrable; given this instruction, it is not at all obvious what it does— and even after you figure that out, it is not obvious, how to change the result register to be register 12.

In the meanwhile, however, the MIPS assembly instruction for the same operation is:

```
add $16, $20, $17
```

This is much more readable— without knowing anything whatsoever about MIPS assembly language, from the `add` it seems likely that addition is somehow involved, and the operands of the addition are somehow related to the numbers 16, 20, and 17. A scan through the tables in the next chapter of this book confirms that `add` performs addition, and that the first operand is the register in which to put the sum of the registers indicated by the second and third operands. At this point, it is clear how to change the result register to 12!

### 7.2 Getting Started: add.asm

To get our feet wet, we’ll write an assembly language program named `add.asm` that computes the sum of 1 and 2, and stores the result in register $t0.

#### 7.2.1 Commenting

Before we start to write the executable statements of program, however, we’ll need to write a comment that describes what the program is supposed to do. In the MIPS assembly language, any text between a pound sign (`#`) and the subsequent newline is considered to be a comment. Comments are absolutely essential! Assembly language programs are notoriously difficult to read unless they are properly documented. Therefore, we start by writing the following:

```
# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2, leaving the result in register $t0.
# Registers used:
# t0 -- used to hold the result.
#
# end of add.asm
```

Even though this program doesn’t actually do anything yet, at least anyone reading our program will know what this program is supposed to do, and who to blame if it doesn’t work. We are not

---

2You should put your own name on your own programs, of course; Dan Ellard shouldn’t take all the blame.
finished commenting this program, but we’ve done all that we can do until we know a little more about how the program will actually work.

7.2.2 Finding the Right Instructions

Next, we need to figure out what instructions the computer will need to execute in order to add two numbers. Since the MIPS architecture has relatively few instructions, it won’t be long before you have memorized all of the instructions that you’ll need, but as you are getting started you’ll need to spend some time browsing through the lists of instructions, looking for ones that you can use to do what you want. Documentation for the MIPS instruction set can be found in chapter 8 of this document.

Luckily, as we look through the list of arithmetic instructions, we notice the `add` instruction, which adds two numbers together.

The `add` operation takes three operands:

1. A register that will be used to store the result of the addition. For our program, this will be `$t0$).

2. A register which contains the first number to be added.

   Therefore, we’re going to have to get 1 into a register before we can use it as an operand of `add`. Checking the list of registers used by this program (which is an essential part of the commenting) we select `$t1$`, and make note of this in the comments.

3. A register which holds the second number, or a 32-bit constant. In this case, since 2 is a constant that fits in 32 bits, we can just use 2 as the third operand of `add`.

We now know how we can add the numbers, but we have to figure out how to get 1 into register `$t1$`. To do this, we can use the `li` (load immediate value) instruction, which loads a 32-bit constant into a register. Therefore, we arrive at the following sequence of instructions:

```
# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2, leaving the result in register $t0$.
# Registers used:
#  t0    - used to hold the result.
#  t1    - used to hold the constant 1.
    li  $t1, 1       # load 1 into $t1.
    add $t0, $t1, 2  # $t0 = $t1 + 2.
```
7.2.3 Completing the Program

These two instructions perform the calculation that we want, but they do not form a complete program. Much like C, an assembly language program must contain some additional information that tells the assembler where the program begins and ends.

The exact form of this information varies from assembler to assembler (note that there may be more than one assembler for a given architecture, and there are several for the MIPS architecture). This tutorial will assume that SPIM is being used as the assembler and runtime environment.

7.2.3.1 Labels and main

To begin with, we need to tell the assembler where the program starts. In SPIM, program execution begins at the location with the label main. A label is a symbolic name for an address in memory. In MIPS assembly, a label is a symbol name (following the same conventions as C symbol names), followed by a colon. Labels must be the first item on a line. A location in memory may have more than one label. Therefore, to tell SPIM that it should assign the label main to the first instruction of our program, we could write the following:

```
# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0 -- used to hold the result.
# t1 -- used to hold the constant 1.
main: li $t1, 1 # load 1 into $t1.
      add $t0, $t1, 2 # $t0 = $t1 + 2.
```

When a label appears alone on a line, it refers to the following memory location. Therefore, we could also write this with the label main on its own line. This is often much better style, since it allows the use of long, descriptive labels without disrupting the indentation of the program. It also leaves plenty of space on the line for the programmer to write a comment describing what the label is used for, which is very important since even relatively short assembly language programs may have a large number of labels.

Note that the SPIM assembler does not permit the names of instructions to be used as labels. Therefore, a label named add is not allowed, since there is an instruction of the same name. (Of course, since the instruction names are all very short and fairly general, they don’t make very descriptive label names anyway.)

Giving the main label its own line (and its own comment) results in the following program:
7.2. GETTING STARTED: ADD.ASM

# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
#    t0 - used to hold the result.
#    t1 - used to hold the constant 1.

main: # SPIM starts execution at main.
    li $t1, 1 # load 1 into $t1.
    add $t0, $t1, 2 # $t0 = $t1 + 2.

# end of add.asm

7.2.3.2 Syscalls

The end of a program is defined in a very different way. Similar to C, where the exit function can be called in order to halt the execution of a program, one way to halt a MIPS program is with something analogous to calling exit in C. Unlike C, however, if you forget to “call exit” your program will not gracefully exit when it reaches the end of the main function. Instead, it will blunder on through memory, interpreting whatever it finds as instructions to execute. Generally speaking, this means that if you are lucky, your program will crash immediately; if you are unlucky, it will do something random and then crash.

The way to tell SPIM that it should stop executing your program, and also to do a number of other useful things, is with a special instruction called a syscall. The syscall instruction suspends the execution of your program and transfers control to the operating system. The operating system then looks at the contents of register $v0 to determine what it is that your program is asking it to do.

Note that SPIM sycalls are not real sycalls; they don’t actually transfer control to the UNIX operating system. Instead, they transfer control to a very simple simulated operating system that is part of the SPIM program.

In this case, what we want is for the operating system to do whatever is necessary to exit our program. Looking in table 8.6.1, we see that this is done by placing a 10 (the number for the exit syscall) into $v0 before executing the syscall instruction. We can use the li instruction again in order to do this:

# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.

3You can “return” from main, just as you can in C, if you treat main as a function. See section 7.10 for more information.
CHAPTER 7. A MIPS TUTORIAL

# Registers used:
# t0 - used to hold the result.
# t1 - used to hold the constant 1.
# v0 - syscall parameter.

main: # SPIM starts execution at main.
    li $t1, 1 # load 1 into $t1.
    add $t0, $t1, 2 # compute the sum of $t1 and 2, and
    # put it into $t0.
    li $v0, 10 # syscall code 10 is for exit.
    syscall # make the syscall.

# end of add.asm

7.3 Using SPIM

At this point, we should have a working program. Now, it’s time to try running it to see what happens.

To run SPIM, simply enter the command `spim` at the commandline. SPIM will print out a message similar to the following:

```
% spim
SPIM Version 5.4 of Jan. 17, 1994
Copyright 1990-1994 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README a full copyright notice.
Loaded: /home/usr6/cs51/de51/SPIM/lib/trap.handler
```

Whenever you see the `(spim)` prompt, you know that SPIM is ready to execute a command.

In this case, since we want to run the program that we just wrote, the first thing we need to do is `load` the file containing the program. This is done with the `load` command:

```
(spim) load "add.asm"
```

The `load` command reads and assembles a file containing MIPS assembly language, and then loads it into the SPIM memory. If there are any errors during the assembly, error messages with line number are displayed. You should not try to execute a file that has not loaded successfully—SPIM will let you run the program, but it is unlikely that it will actually work.

Once the program is loaded, you can use the `run` command to execute it:

---

4The exact text will be different on different computers.
7.4. Using syscall: add2.asm

The program runs, and then SPIM indicates that it is ready to execute another command. Since our program is supposed to leave its result in register $t0, we can verify that the program is working by asking SPIM to print out the contents of $t0, using the print command, to see if it contains the result we expect:

(spm) print $t0
Reg 8 = 0x00000003 (3)

The print command displays the register number followed by its contents in both hexadecimal and decimal notation. Note that SPIM automatically translates from the symbolic name for the register (in this case, $t0) to the actual register number (in this case, $8).

7.4 Using syscall: add2.asm

Our program to compute $1 + 2$ is not particularly useful, although it does demonstrate a number of important details about programming in MIPS assembly language and the SPIM environment. For our next example, we’ll write a program named add2.asm that computes the sum of two numbers specified by the user at runtime, and displays the result on the screen.

The algorithm this program will follow is:

1. Read the two numbers from the user.
   We’ll need two registers to hold these two numbers. We can use $t0$ and $t1$ for this.

2. Compute their sum.
   We’ll need a register to hold the result of this addition. We can use $t2$ for this.

3. Print the sum.

4. Exit. We already know how to do this, using syscall.

Once again, we start by writing a comment. From what we’ve learned from writing add.asm, we actually know a lot about what we need to do; the rest we’ll only comment for now:

# Daniel J. Ellard -- 02/21/94
# add2.asm-- A program that computes and prints the sum
# of two numbers specified at runtime by the user.
# Registers used:
# $t0 -- used to hold the first number.
CHAPTER 7. A MIPS TUTORIAL

# $t1 - used to hold the second number.
# $t2 - used to hold the sum of the $t1 and $t2.
# $v0 - syscall parameter.

main:
## Get first number from user, put into $t0.
## Get second number from user, put into $t1.
add $t2, $t0, $t1 # compute the sum.
## Print out $t2.
li $v0, 10 # syscall code 10 is for exit.
syscall # make the syscall.

# end of add2.asm.

7.4.1 Reading and Printing Integers

The only parts of the algorithm that we don’t know how to do yet are to read the numbers from the user, and print out the sum. Luckily, both of these operations can be done with a syscall. Looking again in table 8.6.1, we see that syscall 5 can be used to read an integer into register $v0, and and syscall 1 can be used to print out the integer stored in $a0.

The syscall to read an integer leaves the result in register $v0, however, which is a small problem, since we want to put the first number into $t0 and the second into $t1. Luckily, in section 8.4.4.3 we find the move instruction, which copies the contents of one register into another.

Note that there are good reasons why we need to get the numbers out of $v0 and move them into other registers: first, since we need to read in two integers, we’ll need to make a copy of the first number so that when we read in the second number, the first isn’t lost. In addition, when reading through the register use guidelines (in section 8.3), we see that register $v0 is not a recommended place to keep anything, so we know that we shouldn’t leave the second number in $v0 either.

This gives the following program:

# Daniel J. Ellard -- 02/21/94
# add2.asm -- A program that computes and prints the sum
# of two numbers specified at runtime by the user.
# Registers used:
# $t0 - used to hold the first number.
# $t1 - used to hold the second number.
# $t2 - used to hold the sum of the $t1 and $t2.
# $v0 - syscall parameter and return value.
# $a0 - syscall parameter.
main:
    ## Get first number from user, put into $t0.
    li $v0, 5  # load syscall read_int into $v0.
    syscall  # make the syscall.
    move $t0, $v0  # move the number read into $t0.

    ## Get second number from user, put into $t1.
    li $v0, 5  # load syscall read_int into $v0.
    syscall  # make the syscall.
    move $t1, $v0  # move the number read into $t1.

    add $t2, $t0, $t1  # compute the sum.

    ## Print out $t2.
    move $a0, $t2  # move the number to print into $a0.
    li $v0, 1  # load syscall print_int into $v0.
    syscall  # make the syscall.

    li $v0, 10  # syscall code 10 is for exit.
    syscall  # make the syscall.

# end of add2.asm.

7.5 Strings: the hello Program

The next program that we will write is the “Hello World” program. Looking in table 8.6.1 once again, we note that there is a syscall to print out a string. All we need to do is to put the address of the string we want to print into register $a0, the constant 4 into $v0, and execute syscall. The only things that we don’t know how to do are how to define a string, and then how to determine its address.

The string "Hello World" should not be part of the executable part of the program (which contains all of the instructions to execute), which is called the text segment of the program. Instead, the string should be part of the data used by the program, which is, by convention, stored in the data segment. The MIPS assembler allows the programmer to specify which segment to store each item in a program by the use of several assembler directives. (see 8.5.1 for more information)

To put something in the data segment, all we need to do is to put a .data before we define it. Everything between a .data directive and the next .text directive (or the end of the file) is put into the data segment. Note that by default, the assembler starts in the text segment, which is why our earlier programs worked properly even though we didn’t explicitly mention which segment to use. In general, however, it is a good idea to include segment directives in your code, and we will
do so from this point on.

We also need to know how to allocate space for and define a null-terminated string. In the 
MIPS assembler, this can be done with the `.asciiz` (ASCII, zero terminated string) directive. 
For a string that is not null-terminated, the `.ascii` directive can be used (see 8.5.2 for more 
information).

Therefore, the following program will fulfill our requirements:

```
# Daniel J. Ellard -- 02/21/94
# hello.asm-- A "Hello World" program.
# Registers used:
# $v0 - syscall parameter and return value.
# $a0 - syscall parameter-- the string to print.

.text
main:
  la $a0, hello_msg # load the addr of hello_msg into $a0.
  li $v0, 4 # 4 is the print_string syscall.
  syscall # do the syscall.
  li $v0, 10 # 10 is the exit syscall.
  syscall # do the syscall.

# Data for the program:
.data
hello_msg: .asciiz "Hello World\n"
```

Note that data in the data segment is assembled into adjacent locations. Therefore, there are 
many ways that we could have declared the string "Hello World\n" and gotten the same exact 
output. For example we could have written our string as:

```
.data
hello_msg:   .ascii  "Hello" # The word "Hello"
             .ascii  " "   # the space.
             .ascii  "World" # The word "World"
             .ascii  "\n"   # A newline.
             .byte  0      # a 0 byte.
```

If we were in a particularly cryptic mood, we could have also written it as:

```
.data
hello_msg:   .byte  0x48 # hex for ASCII "H"
             .byte  0x65 # hex for ASCII "e"
```
7.6. CONDITIONAL EXECUTION: THE LARGER PROGRAM

You can use the .data and .text directives to organize the code and data in your programs in whatever is most stylistically appropriate. The example programs generally have the all of the .data items defined at the end of the program, but this is not necessary. For example, the following code will assemble to exactly the same program as our original hello.asm:

```
.text # put things into the text segment...
main:
.data # put things into the data segment...
hello_msg: .asciiz "Hello World\n"
.text # put things into the text segment...
la $a0, hello_msg # load the addr of hello_msg into $a0.
li $v0, 4 # 4 is the print_string syscall.
syscall # do the syscall.
li $v0, 10 # 10 is the exit syscall.
syscall # do the syscall.
```

7.6 Conditional Execution: the larger Program

The next program that we will write will explore the problems of implementing conditional execution in MIPS assembler language. The actual program that we will write will read two numbers from the user, and print out the larger of the two.

One possible algorithm for this program is exactly the same as the one used by add2.asm, except that we’re computing the maximum rather than the sum of two numbers. Therefore, we’ll start by copying add2.asm, but replacing the add instruction with a placeholder comment:

```
# Daniel J. Ellard -- 02/21/94
# larger.asm-- prints the larger of two numbers specified at runtime by the user.
# Registers used:
#  $t0  - used to hold the first number.
#  $t1  - used to hold the second number.
#  $t2  - used to store the larger of $t1 and $t2.

.text
```
main:
    ## Get first number from user, put into $t0.
    li   $v0, 5    # load syscall read_int into $v0.
syscall    # make the syscall.
move   $t0, $v0    # move the number read into $t0.

    ## Get second number from user, put into $t1.
    li   $v0, 5    # load syscall read_int into $v0.
syscall    # make the syscall.
move   $t1, $v0    # move the number read into $t1.

    ## put the larger of $t0 and $t1 into $t2.
    ## (placeholder comment)

    ## Print out $t2.
move   $a0, $t2    # move the number to print into $a0.
    li   $v0, 1    # load syscall print_int into $v0.
syscall    # make the syscall.

    ## exit the program.
    li   $v0, 10    # syscall code 10 is for exit.
syscall    # make the syscall.

# end of larger.asm.

Browsing through the instruction set again, we find in section 8.4.3.1 a description of the MIPS branching instructions. These allow the programmer to specify that execution should branch (or jump) to a location other than the next instruction. These instructions allow conditional execution to be implemented in assembler language (although in not nearly as clean a manner as higher-level languages provide).

One of the branching instructions is \texttt{bgt}. The \texttt{bgt} instruction takes three arguments. The first two are numbers, and the last is a label. If the first number is larger than the second, then execution should continue at the label, otherwise it continues at the next instruction. The \texttt{b} instruction, on the other hand, simply branches to the given label.

These two instructions will allow us to do what we want. For example, we could replace the placeholder comment with the following:

\begin{verbatim}
# If \$t0 > \$t1, branch to \$t0_bigger,
bgt      \$t0, \$t1, \$t0_bigger
move      \$t2, \$t1        # otherwise, copy \$t1 into \$t2.
b     \$t0_bigger:
move      \$t2, \$t0        # and then branch to endif

# copy \$t0 into \$t2
endif:
\end{verbatim}
If $t0$ is larger, then execution will branch to the $t0_bigger$ label, where $t0$ will be copied to $t2$. If it is not, then the next instructions, which copy $t1$ into $t2$ and then branch to the endif label, will be executed.

This gives us the following program:

```assembly
.text
main:
    ## Get first number from user, put into $t0.
    li $v0, 5  # load syscall read_int into $v0.
    syscall  # make the syscall.
    move $t0, $v0  # move the number read into $t0.

    ## Get second number from user, put into $t1.
    li $v0, 5  # load syscall read_int into $v0.
    syscall  # make the syscall.
    move $t1, $v0  # move the number read into $t1.

    ## put the larger of $t0 and $t1 into $t2.
    bgt $t0, $t1, t0_bigger  # If $t0 > $t1, branch to t0_bigger,
    move $t2, $t1  # otherwise, copy $t1 into $t2.
    b endif  # and then branch to endif

    t0_bigger:
    move $t2, $t0  # copy $t0 into $t2

    endif:
    ## Print out $t2.
    move $a0, $t2  # move the number to print into $a0.
    li $v0, 1  # load syscall print_int into $v0.
    syscall  # make the syscall.

    ## exit the program.
    li $v0, 10  # syscall code 10 is for exit.
    syscall  # make the syscall.

# end of larger.asm.
```
7.7 Looping: the multiples Program

The next program that we will write will read two numbers \( A \) and \( B \), and print out multiples of \( A \) from \( A \) to \( A \times B \). The algorithm that our program will use is given in algorithm 7.1. This algorithm translates easily into MIPS assembly. Since we already know how to read in numbers and print them out, we won’t bother to implement these steps here— we’ll just leave these as comments for now.

**Algorithm 7.1** The multiples program.

1. Get \( A \) from the user.
2. Get \( B \) from the user. If \( B \leq 0 \), terminate.
3. Set sentinel value \( S = A \times B \).
4. Set multiple \( m = A \).
5. Loop:
   (a) Print \( m \).
   (b) If \( m == S \), then go to the next step.
   (c) Otherwise, set \( m = m + A \), and then repeat the loop.
6. Terminate.

```
# Daniel J. Ellard -- 02/21/94
# multiples.asm-- takes two numbers A and B, and prints out
# all the multiples of A from A to A * B.
# If B <= 0, then no multiples are printed.
# Registers used:
# $t0 -- used to hold A.
# $t1 -- used to hold B.
# $t2 -- used to store S, the sentinel value A * B.
# $t3 -- used to store m, the current multiple of A.
.text
main:
    ## read A into $t0, B into $t1 (omitted).
```
blez $t1, exit            # if B <= 0, exit.
mul $t2, $t0, $t1         # S = A * B.
move $t3, $t0             # m = A

loop:
    ## print out $t3 (omitted)
    beq $t2, $t3, endloop    # if m == S, we’re done.
    add $t3, $t3, $t0        # otherwise, m = m + A.
    ## print a space (omitted)
    b loop

endloop:
    ## exit (omitted)
# end of multiples.asm

The complete code for this program is listed in section 9.3.

7.8  Loads: the palindromes.asm Program

The next program that we write will read a line of text and determine whether or not the text is a palindrome. A palindrome is a word or sentence that spells exactly the same thing both forward and backward. For example, the string “anna” is a palindrome, while “ann” is not. The algorithm that we’ll be using to determine whether or not a string is a palindrome is given in algorithm 7.2.

Note that in the more common definition of a palindrome, whitespace, capitalization, and punctuation are ignored, so the string “Able was I ere I saw Elba.” would be considered a palindrome, but by our definition it is not. (In exercise 7.12.2, you get to fix this oversight.)

Once again, we start with a comment:

## Daniel J. Ellard -- 02/21/94
## palindrome.asm -- reads a line of text and tests if it is a palindrome.
## Register usage:
##   # $t1 - A.
##   # $t2 - B.
##   # $t3 - the character at address A.
##   # $t4 - the character at address B.
##   # $v0 - syscall parameter / return values.
##   # $a0 - syscall parameters.
##   # $a1 - syscall parameters.
Algorithm 7.2 To determine if the string that starts at address $S$ is a palindrome.
This algorithm is appropriate for the strings that end with a newline followed by a 0 character, as strings read in by the read_string syscall do. (See exercise 7.12.1 to generalize this algorithm.)
Note that in this algorithm, the operation of getting the character located at address $X$ is written as $*X$.

1. Let $A = S$.

2. Let $B =$ a pointer to the last character of $S$. To find the last character in $S$, use the following algorithm:
   
   (a) Let $B = S$.
   (b) Loop:
       
       - If $*B == 0$ (i.e. the character at address $B$ is 0), then $B$ has gone past the end of the string. Set $B = B - 2$ (to move $B$ back past the 0 and the newline), and continue with the next step.
       - Otherwise, set $B = (B + 1)$.

3. Loop:
   
   (a) If $A \geq B$, then the string is a palindrome. Halt.
   (b) If $*A \neq *B$, then the string is not a palindrome. Halt.
   (c) Set $A = (A + 1)$.
   (d) Set $B = (B - 1)$.
7.8. LOADS: THE PALINDROME.ASM PROGRAM

The first step of the algorithm is to read in the string from the user. This can be done with the `read_string` syscall (syscall number 8), which is similar in function to the `fgets` function in the C standard I/O library. To use this syscall, we need to load into register $a0 the pointer to the start of the memory that we have set aside to hold the string. We also need to load into register $a1 the maximum number of bytes to read.

To set aside the space that we’ll need to store the string, the `.space` directive can be used. This gives the following code:

```
.text
main: # SPIM starts by jumping to main.
    ## read the string S:
    la $a0, string_space
    li $a1, 1024
    li $v0, 8 # load "read_string" code into $v0.
    syscall

.string_space: .space 1024 # set aside 1024 bytes for the string.
```

Once we’ve got the string, then we can use algorithm 7.2 (on page 232). The first step is simple enough; all we need to do is load the address of `string_space` into register `$t1`, the register that we’ve set aside to represent $A$:

```
la $t1, string_space # A = S.
```

The second step is more complicated. In order to compare the character pointed to by $B$ with 0, we need to load this character into a register. This can be done with the `lb` (load byte) instruction:

```
la $t2, string_space ## we need to move B to the end
length_loop: # of the string:
    lb $t3, ($t2) # load the byte at B into $t3.
    beqz $t3, end_length_loop # if $t3 == 0, branch out of loop.
    addu $t2, $t2, 1 # otherwise, increment B,
    b length_loop # and repeat
end_length_loop:
    subu $t2, $t2, 2 ## subtract 2 to move B back past
                  # the '\0' and '\n'.
```

Note that the arithmetic done on the pointer $B$ is done using unsigned arithmetic (using `addu` and `subu`). Since there is no way to know where in memory a pointer will point, the numerical value of the pointer may well be a “negative” number if it is treated as a signed binary number.

When this step is finished, $A$ points to the first character of the string and $B$ points to the last. The next step determines whether or not the string is a palindrome:
CHAPTER 7. A MIPS TUTORIAL

test_loop:
    bge $t1, $t2, is_palin  # if $A \geq B$, it's a palindrome.
    lb $t3, (\$t1)  # load the byte at address $A$ into $t3,
    lb $t4, (\$t2)  # load the byte at address $B$ into $t4.
    bne $t3, $t4, not_palin  # if $t3 \neq t4$, not a palindrome.
    addu $t1, $t1, 1  # increment $A$,
    subu $t2, $t2, 1  # decrement $B$,
    b test_loop  # and repeat the loop.

# Otherwise,

The complete code for this program is listed in section 9.4 (on page 266).

7.9 The atoi Program

The next program that we'll write will read a line of text from the terminal, interpret it as an integer, and then print it out. In effect, we'll be reimplementing the read_int system call (which is similar to the GetInteger function in the Roberts libraries).

7.9.1 atoi-1

We already know how to read a string, and how to print out a number, so all we need is an algorithm to convert a string into a number. We'll start with the algorithm given in 7.3 (on page 235).

Let's assume that we can use register $t0$ as $S$, register $t2$ as $D$, and register $t1$ is available as scratch space. The code for this algorithm then is simply:

li $t2, 0  # Initialize sum = 0.
sum_loop:
    lb $t1, (\$t0)  # load the byte *S into $t1,
    addu $t0, $t0, 1  # and increment S.
    ## use 10 instead of \n due to SPIM bug!
    beq $t1, 10, end_sum_loop  # if $t1 == \n, branch out of loop.
    mul $t2, $t2, 10  # $t2 *= 10.
    sub $t1, $t1, 0  # $t1 -= 0.'
    add $t2, $t2, $t1  # $t2 += $t1.
    b sum_loop  # and repeat the loop.
end_sum_loop:

The complete code for this program is listed in section 9.4 (on page 266).
Algorithm 7.3 To convert an ASCII string representation of a integer into the corresponding integer.
Note that in this algorithm, the operation of getting the character at address $X$ is written as $\ast X$.

- Let $S$ be a pointer to start of the string.
- Let $D$ be the number.

1. Set $D = 0$.

2. Loop:
   
   (a) If $\ast S = \ 'n'$, then continue with the next step.
   
   (b) Otherwise,
   
   i. $S = (S + 1)$
   
   ii. $D = (D \times 10)$
   
   iii. $D = (D + (\ast S - '0'))$

   In this step, we can take advantage of the fact that ASCII puts the numbers with represent the digits 0 through 9 are arranged consecutively, starting at 0. Therefore, for any ASCII character $x$, the number represented by $x$ is simply $x - '0'$.
CHAPTER 7. A MIPS TUTORIAL

Note that due to a bug in the SPIM assembler, the \texttt{beq} must be given the constant 10 (which is the ASCII code for a newline) rather than the symbolic character code ‘\n’, as you would use in C. The symbol ‘\n’ does work properly in strings declarations (as we saw in the \texttt{hello.asm} program).

A complete program that uses this code is in \texttt{atoi-1.asm}.

\textbf{7.9.2 \texttt{atoi-2}}

Although the algorithm used by \texttt{atoi-1} seems reasonable, it actually has several problems. The first problem is that this routine cannot handle negative numbers. We can fix this easily enough by looking at the very first character in the string, and doing something special if it is a ‘\texttt{-}’. The easiest thing to do is to introduce a new variable, which we’ll store in register $t3$, which represents the sign of the number. If the number is positive, then $t3$ will be 1, and if negative then $t3$ will be -1. This makes it possible to leave the rest of the algorithm intact, and then simply multiply the result by $t3$ in order to get the correct sign on the result at the end:

\begin{verbatim}
li $t2, 0  # Initialize sum = 0.
get_sign:
    li $t3, 1
    lb $t1, ($t0)  # grab the "sign"
    bne $t1, '-', positive  # if not "-", do nothing.
    li $t3, -1  # otherwise, set t3 = -1, and
    addu $t0, $t0, 1  # skip over the sign.
positive:
sum_loop:  ## sum_loop is the same as before.
end_sum_loop:
    mul $t2, $t2, $t3  # set the sign properly.
\end{verbatim}

A complete program that incorporates these changes is in \texttt{atoi-2.asm}.

\textbf{7.9.3 \texttt{atoi-3}}

While the algorithm in \texttt{atoi-2.asm} is better than the one used by \texttt{atoi-1.asm}, it is by no means free of bugs. The next problem that we must consider is what happens when $S$ does not point to a proper string of digits, but instead points to a string that contains erroneous characters.

If we want to mimic the behavior of the UNIX \texttt{atoi} library function, then as soon as we encounter any character that isn’t a digit (after an optional ‘\texttt{-}’) then we should stop the conversion
immediately and return whatever is in \( D \) as the result. In order to implement this, all we need to do is add some extra conditions to test on every character that gets read in inside \texttt{sum_loop}:

\begin{verbatim}
sum_loop:
    lb   $t1, ($t0)   # load the byte \( \ast S \) into \( t1 \),
    addu $t0, $t0, 1  # and increment \( S \),

    ## use 10 instead of \texttt{\'\textbackslash n\textbackslash n\textbackslash n\textbackslash n\textbackslash n}' due to SPIM bug!
    beq   $t1, 10, end_sum_loop  # if \( t1 == \text{\textbackslash n} \), branch out of loop.
    blt   $t1, '0', end_sum_loop  # make sure \( 0 \leq t1 \)
    bgt   $t1, '9', end_sum_loop  # make sure \( 9 \geq t1 \)
    mul  $t2, $t2, 10        # \( t2 *= 10 \).
    sub  $t1, $t1, '0'       # \( t1 -= '0' \).
    add  $t2, $t2, $t1       # \( t2 += t1 \).

    b     sum_loop          # and repeat the loop.
end_sum_loop:
\end{verbatim}

A complete program that incorporates these changes is in \texttt{atoi-3.asm}.

\subsection*{7.9.4 \texttt{atoi-4}}

While the algorithm in \texttt{atoi-3.asm} is nearly correct (and is at least as correct as the one used by the standard \texttt{atoi} function), it still has an important bug. The problem is that algorithm 7.3 (and the modifications we’ve made to it in \texttt{atoi-2.asm} and \texttt{atoi-3.asm}) is generalized to work with \textit{any} number. Unfortunately, register \$t2, which we use to represent \( D \), can only represent 32-bit binary number. Although there’s not much that we can do to \textit{prevent} this problem, we definitely want to \textit{detect} this problem and indicate that an error has occurred.

There are two spots in our routine where an overflow might occur: when we multiply the contents of register \$t2 by 10, and when we add in the value represented by the current character. Detecting overflow during multiplication is not hard. Luckily, in the MIPS architecture, when multiplication and division are performed, the result is actually stored in two 32-bit registers, named \texttt{lo} and \texttt{hi}. For division, the quotient is stored in \texttt{lo} and the remainder in \texttt{hi}. For multiplication, \texttt{lo} contains the low-order 32 bits and \texttt{hi} contains the high-order 32 bits of the result. Therefore, if \texttt{hi} is non-zero after we do the multiplication, then the result of the multiplication is too large to fit into a single 32-bit word, and we can detect the error.

We’ll use the \texttt{mul} instruction to do the multiplication, and then the \texttt{mfhi} (move from \texttt{hi}) and \texttt{mflo} (move from \texttt{lo}) instructions to get the results.
To implement this we need to replace the single line that we used to use to do the multiplication with the following:

```
# Note-- $t4 contains the constant 10.
mult $t2, $t4 # multiply $t2 by 10.
mfhi $t5 # check for overflow;
bnez $t5, overflow # if so, then report an overflow.
mflo $t2 # get the result of the multiply
```

There’s another error that can occur here, however: if the multiplication makes the number too large to be represented as a positive two’s complement number, but not quite large enough to require more than 32 bits. (For example, the number 3000000000 will be converted to -1294967296 by our current routine.) To detect whether or not this has happened, we need to check whether or not the number in register $t2 appears to be negative, and if so, indicate an error. This can be done by adding the following instruction immediately after the mflo:

```
blt $t2, $0, overflow # make sure that it isn’t negative.
```

This takes care of checking that the multiplication didn’t overflow. We can detect whether an addition overflowed in much the same manner, by adding the same test immediately after the addition.

The resulting code, along with the rest of the program, can be found in section 9.6 (on page 270).

### 7.10 Function Environments and Linkage

One of the most important benefits of a high-level language such as C is the notion of a function. In C, a function provides several useful abstractions:

- The mapping of actual parameters to formal parameters.
- Allocation and initialization of temporary local storage. This is particularly important in languages which allow recursion: each call to the function must get its own copy of any local variables, to prevent one call to a recursive function from clobbering the values of a surrounding call to the same function.

The information that describes the state of a function during execution (i.e. the actual parameters, the value of all of the local variables, and which statement is being executed) is called the environment of the function. (Note that the values of any global variables referenced by the function are not part of the environment.) For a MIPS assembly program, the environment of a function consists of the values of all of the registers that are referenced in the function (see exercise 7.12.3).
In order to implement the ability to save and restore a function’s environment, most architectures, including the MIPS, use the stack to store each of the environments.

In general, before a function $A$ calls function $B$, it pushes its environment onto the stack, and then jumps to function $B$. When the function $B$ returns, function $A$ restores its environment by popping it from the stack. In the MIPS software architecture, this is accomplished with the following procedure:

1. The **caller** must:
   
   (a) Put the parameters into $a0$-$a3$. If there are more than four parameters, the additional parameters are pushed onto the stack.
   
   (b) Save any of the **caller-saved** registers ($t0$-$t9$) which are used by the caller.
   
   (c) Execute a `jal` (or `jalr`) to jump to the function.

2. The **callee** must, as part of the function preamble:
   
   (a) Create a stack frame, by subtracting the frame size from the stack pointer ($sp$).

   Note that the minimum stack frame size in the MIPS software architecture is 32 bytes, so even if you don’t need all of this space, you should still make your stack frames this large.

   (b) Save any callee-saved registers ($s0$-$s7$, $fp$, $ra$) which are used by the callee.

   Note that the frame pointer ($fp$) must always be saved. The return address ($ra$) needs to be saved only by functions which make function calls themselves.

   (c) Set the frame pointer to the stack pointer, plus the frame size.

3. The **callee** then executes the body of the function.

4. To return from a function, the **callee** must:
   
   (a) Put the return value, if any, into register $v0$.

   (b) Restore callee-saved registers.

   (c) Jump back to $ra$, using the `jr` instruction.

5. To clean up after a function call, the **caller** must:
   
   (a) Restore the caller-saved registers.

   (b) If any arguments were passed on the stack (instead of in $a0$-$a3$), pop them off of the stack.
(c) Extract the return value, if any, from register $v0.

The convention used by the programs in this document is that a function stores $fp at the top of its stack frame, followed by $ra, then any of the callee-saved registers ($s0 - $s7), and finally any of the caller-saved registers ($t0 - $t9) that need to be preserved.

### 7.10.1 Computing Fibonacci Numbers

The Fibonacci sequence has the following recursive definition: let $F(n)$ be the $n$th element (where $n \geq 0$) in the sequence:

- If $n < 2$, then $F(n) \equiv 1$. (*the base case*)
- Otherwise, $F(n) = F(n - 1) + F(n - 2)$. (*the recursive case*)

This definition leads directly to a recursive algorithm for computing the $n$th Fibonacci number. As you may have realized, particularly if you’ve seen this sequence before, there are much more efficient ways to compute the $n$th Fibonacci number. Nevertheless, this algorithm is often used to demonstrate recursion—so here we go again.

In order to demonstrate a few different aspects of the MIPS function calling conventions, however, we’ll implement the `fib` function in a few different ways.

#### 7.10.1.1 Using Saved Registers: fib-s.asm

The first way that we’ll code this will use callee-saved registers to hold all of the local variables.

```assembly
# fib-- (callee-save method)
# Registers used:
# $a0 - initially n.
# $s0 - parameter n.
# $s1 - fib (n - 1).
# $s2 - fib (n - 2).
.text
.fib:
  subu $sp, $sp, 32               # frame size = 32, just because...
  sw $ra, 28($sp)                 # preserve the Return Address.
  sw $fp, 24($sp)                 # preserve the Frame Pointer.
  sw $s0, 20($sp)                 # preserve $s0.
  sw $s1, 16($sp)                 # preserve $s1.
  sw $s2, 12($sp)                 # preserve $s2.
  addu $fp, $sp, 32               # move Frame Pointer to base of frame.
```
move $s0, $a0  # get n from caller.
blt $s0, 2, fib_base_case  # if n < 2, then do base case.
sub $a0, $s0, 1  # compute fib (n - 1)
jal fib  #
move $s1, $v0  # s1 = fib (n - 1).
sub $a0, $s0, 2  # compute fib (n - 2)
jal fib
move $s2, $v0  # $s2 = fib (n - 2).
add $v0, $s1, $s2  # $v0 = fib (n - 1) + fib (n - 2).
b fib_return

fib_base_case:  # in the base case, return 1.
    li $v0, 1

fib_return:
lw $ra, 28($sp)  # restore the Return Address.
lw $fp, 24($sp)  # restore the Frame Pointer.
lw $s0, 20($sp)  # restore $s0.
lw $s1, 16($sp)  # restore $s1.
lw $s2, 12($sp)  # restore $s2.
addu $sp, $sp, 32  # restore the Stack Pointer.
jr $ra  # return.

As a baseline test, let’s time the execution of this program computing the $F(20)$:

% echo 20 | /bin/time spim -file fib-s.asm
SPIM Version 5.4 of Jan. 17, 1994
Copyright 1990-1994 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README a full copyright notice.
Loaded: /home/usr6/cs51/de51/SPIM/lib/trap.handler
10946
 5.1 real  4.8 user  0.2 sys

7.10.1.2  Using Temporary Registers: fib-t.asm

If you trace through the execution of the fib function in fib-s.asm, you’ll see that roughly half of the function calls are leaf calls. Therefore, it is often unnecessary to go to all of the work of saving all of the registers in each call to fib, since half the time fib doesn’t call itself again. We can take advantage of this fact by using caller saved registers (in this case $t0-t2) instead of callee saved registers. Since it is the responsibility of the caller to save these registers, the code gets somewhat rearranged:
# fib-- (caller-save method)
# Registers used:
# $a0 - initially n.
# $t0 - parameter n.
# $t1 - fib (n - 1).
# $t2 - fib (n - 2).

.text

fib:
    subu $sp, $sp, 32 # frame size = 32, just because...
    sw $ra, 28($sp) # preserve the Return Address.
    sw $fp, 24($sp) # preserve the Frame Pointer.
    addu $fp, $sp, 32 # move Frame Pointer to base of frame.

    move $t0, $a0 # get n from caller.

    blt $t0, 2, fib_base_case # if n < 2, then do base case.

    # call function fib (n - 1):
    sw $t0, 20($sp) # save n.
    sub $a0, $t0, 1 # compute fib (n - 1)
    jal fib
    move $t1, $v0 # $t1 = fib (n - 1)
    lw $t0, 20($sp) # restore n.

    # call function fib (n - 2);
    sw $t0, 20($sp) # save n.
    sw $t1, 16($sp) # save $t1.
    sub $a0, $t0, 2 # compute fib (n - 2)
    jal fib
    move $t2, $v0 # $t2 = fib (n - 2)
    lw $t0, 20($sp) # restore n.
    lw $t1, 16($sp) # restore $t1.

    add $v0, $t1, $t2 # $v0 = fib (n - 1) + fib (n - 2).
    b fib_return

fib_base_case:
    # in the base case, return 1.
    li $v0, 1

fib_return:
    lw $ra, 28($sp) # Restore the Return Address.
    lw $fp, 24($sp) # restore the Frame Pointer.
    addu $sp, $sp, 32 # restore the Stack Pointer.
    jr $ra # return.

Once again, we can time the execution of this program in order to see if this change has made
any improvement:

% echo 20 | /bin/time spim -file fib-t.asm
SPIM Version 5.4 of Jan. 17, 1994
Copyright 1990-1994 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README a full copyright notice.
Loaded: /home/usr6/cs51/de51/SPIM/lib/trap.handler
10946
  4.5 real  4.1 user  0.1 sys

In these tests, the user time is what we want to measure, and as we can see, fib-s.asm is approximately 17% slower than fib-t.asm.

7.10.1.3 Optimization: fib-o.asm

Warning! Hacks ahead! There are still more tricks we can try in order to increase the performance of this program. Of course, the best way to increase the performance of this program would be to use a better algorithm, but for now we’ll concentrate on optimizing our assembly implementation of the algorithm we’ve been using.

Starting with the observation that about half the calls to fib have an argument \( n \) of 1 or 0, and therefore do not need to do anything except return a 1, we can simplify the program considerably: this base case doesn’t require building a stack frame, or using any registers except \$a0 and \$v0. Therefore, we can postpone the work of building a stack frame until after we’ve tested to see if we’re going to do the base case.

In addition, we can further trim down the number of instructions that are executed by saving fewer registers. For example, in the second recursive call to fib it is not necessary to preserve \( n \) – we don’t care if it gets clobbered, since it isn’t used anywhere after this call.

```assembly
## fib-- (hacked-up caller-save method)
## Registers used:
## $a0 - initially n.
## $t0 - parameter n.
## $t1 - fib (n - 1).
## $t2 - fib (n - 2).
.text
fib:
  bgt $a0, 1, fib_recurse       # if n < 2, then just return a 1,
  li  $v0, 1                    # don’t bother to build a stack frame.
  jr  $ra

fib_recurse:                   # otherwise, set things up to handle
  subu $sp, $sp, 32            # the recursive case:
  # frame size = 32, just because...
```

```assembly```
sw $ra, 28($sp)  # preserve the Return Address.
sw $fp, 24($sp)  # preserve the Frame Pointer.
addu $fp, $sp, 32  # move Frame Pointer to base of frame.
move $t0, $a0  # get n from caller.

sw $t0, 20($sp)  # preserve n.
sub $a0, $t0, 1  # compute fib (n - 1)
jal fib
move $t1, $v0  # t1 = fib (n - 1)
lw $t0, 20($sp)  # restore n.

sw $t0, 16($sp)  # preserve $t0.
sub $a0, $t0, 2  # compute fib (n - 2)
jal fib
move $t2, $v0  # t2 = fib (n - 2)
lw $t1, 16($sp)  # restore $t1.
add $v0, $t1, $t2  # $v0 = fib (n - 1) + fib (n - 2)
lw $ra, 28($sp)  # restore Return Address.
lw $fp, 24($sp)  # restore Frame Pointer.
addu $sp, $sp, 32  # restore Stack Pointer.
jr $ra  # return.

Let’s time this and see how it compares:
% echo 20 | /bin/time spim -file fib-o.asm

This is clearly much faster. In fact, it’s nearly twice as fast as the original fib-s.asm. This makes sense, since we have eliminated building and destroying about half of the stack frames, and a large percentage of the fib function does nothing but set up and dismantle the stack frame.

Note that the reason that optimizing the base case of the recursion helps so much with this algorithm is because it occurs about half of the time— but this is not characteristic of all recursive algorithms. For example, in a recursive algorithm to compute the factorial of $n$, the recursive case will occur about $n - 1$ times, while the base case will only occur once. Therefore, it makes more sense to optimize the recursive case in that situation.
7.11. **STRUCTURES AND SBRK: THE TREESORT PROGRAM**

There’s still more that can be done, however; see exercise 7.12.5 to pursue this farther. A complete listing of a program that uses this implementation of the fib function can be found in section 9.8 (on page 276).

7.11  **Structures and sbrk: the treesort Program**

Included in section 9.9 of this document is the source code for a SPIM program that reads a list of numbers from the user and prints out the list in ascending order. The input is read one number per line, using the read_int syscall, until the user types in the sentinel value. The sentinel value is currently 0, but can be changed in the code to any 32-bit integer.

The treesort algorithm should be familiar to anyone who has used ordered binary trees. The general algorithm is shown in 7.4.

---

**Algorithm 7.4** The treesort algorithm.

1. Build an ordered binary tree \( T \) containing all the values to be sorted.
2. Do an inorder traversal of \( T \), printing out the values of each node.

---

Since we already have seen how to write functions (including recursive functions), doing the inorder traversal won’t be much of a problem. Building the tree, however, will require several new techniques: we need to learn how to represent structures (in particular the structure of each node), and we need to learn how to dynamically allocate memory, so we can construct binary trees of arbitrary size.

7.11.1  **Representing Structures**

In C, we would use a definition such as the following for our tree node structures:

```c
typedef struct _tree_t {
    int val; /* the value of this node. */
    struct _tree_t *left; /* pointer to the left child. */
    struct _tree_t *right; /* pointer to the right child. */
} tree_t;
```

We’d complete our definition of this representation by specifying that a NULL pointer will be used as the value of the left field when the node does not have a left child, and as the value of right field when the node does not have a right child.
In assembly language, unfortunately, we need to deal with things on a lower level\(^5\). If we take a look at this structure, we note that in the MIPS architecture it will require exactly three words (twelve bytes) to represent this structure: a word to represent the val, another for the left pointer, and the last for the right pointer (in the MIPS R2000 architecture, a pointer is 32 bits in length, so it will fit in a single word. This is not necessarily the case for other architectures, however). Therefore, we can use a three-word chunk of memory to represent a node, as long as we keep track of what each word in the chunk represents. For example,

\[
\begin{align*}
\text{# MIPS assembly:} & & \text{C equivalent:} \\
\text{lw} & \quad $s0, 0($t1) & \quad \# a = \text{foo}->\text{val}; \\
\text{lw} & \quad $s1, 4($t1) & \quad \# b = \text{foo}->\text{left}; \\
\text{lw} & \quad $s2, 8($t1) & \quad \# c = \text{foo}->\text{right}; \\
\text{sw} & \quad $s0, 0($t1) & \quad \# \text{foo}->\text{val} = a; \\
\text{sw} & \quad $s1, 4($t1) & \quad \# \text{foo}->\text{left} = b; \\
\text{sw} & \quad $s2, 8($t1) & \quad \# \text{foo}->\text{right} = c;
\end{align*}
\]

Needless to say, once you choose a representation you must fully comment it in your code. In addition, any functions or routines that depend on the details of a structure representation should mention this fact explicitly, so that if you change the representation later you’ll know exactly which functions you will also need to change.

### 7.11.2 The `sbrk` syscall

Now that we’ve solved the problem of representing structures, we need to solve the problem of how to dynamically allocate them. Luckily, there is a syscall named `sbrk` that can be used to allocate memory (see section 8.6.1).

Unfortunately, `sbrk` behaves much more like its namesake (the UNIX `sbrk` system call) than like `malloc`– it extends the data segment by the number of bytes requested, and then returns the location of the previous end of the data segment (which is the start of the freshly allocated memory). The problem with `sbrk` is that it can only be used to allocate memory, never to give it back.

---

\(^5\)Some assemblers do have features that allow C-like structure definitions. Unfortunately, SPIM is not one of them, so you need to keep track of this information yourself.
7.12 Exercises

7.12.1
In the palindrome algorithm 7.2, the algorithm for moving B to the end of the string is incorrect if the string does not end with a newline.

Fix algorithm 7.2 so that it behaves properly whether or not there is a newline on the end of the string. Once you have fixed the algorithm, fix the code as well.

7.12.2
Modify the palindrome.asm program so that it ignores whitespace, capitalization, and punctuation.

Your program must be able to recognize the following strings as palindromes:

1. "1 2 321"
2. "Madam, I'm Adam."
3. "Able was I, ere I saw Elba."
4. "A man, a plan, a canal– Panama!"
5. "Go hang a salami; I'm a lasagna hog."

7.12.3
In section 7.10, a function’s environment is defined to be the values of all of the registers that are referenced in the function. If we use this definition, we may include more registers than are strictly necessary. Write a more precise definition, which may in some cases include fewer registers.

7.12.4
Write a MIPS assembly language program named fib-iter.asm that asks the user for n, and then computes and prints the nth Fibonacci sequence using an $O(n)$ iterative algorithm.
7.12.5

The fib-o.asm program (shown in 7.10.1.3) is not completely optimized.

1. Find at least one more optimization, and time your resulting program to see if it is faster than fib-o.asm. Call your program fib-o+.asm.

2. Since you know that fib will never call any function other than fib, can you make use of this to optimize the calling convention for this particular function? You should be able discover (at least) two instructions in fib that are not necessary. With some thought, you may be able to find others.

Design a calling convention optimized for the fib program, and write a program named fib-o++.asm that implements it. Time your resulting program and see how much faster it is than fib-o.asm and your fib-o+.asm program.

3. Time the program from question 7.12.4 and compare times with fib-o.asm, fib-o+.asm, and fib-o++.asm. What conclusion do you draw from your results?

7.12.6

Starting with the routine from atoi-4.asm, write a MIPS assembly language function named atoi that behaves in the same manner as the atoi function in the C library. Your function must obey the MIPS calling conventions, so that it can be used in any program. How should your function indicate to its caller that an overflow has occurred?

7.12.7

Write a MIPS assembly language program that asks the user for 20 numbers, bubblesorts them, and then prints them out in ascending order.

7.12.8

Write a MIPS assembly language program that asks the user for 20 numbers, mergesorts them, and then prints them out in ascending order.
Chapter 8

The MIPS R2000 Instruction Set

by Daniel J. Ellard

8.1 A Brief History of RISC

In the beginning of the history of computer programming, there were no high-level languages. All programming was initially done in the native machine language and later the native assembly language of whatever machine was being used.

Unfortunately, assembly language is almost completely nonportable from one architecture to another, so every time a new and better architecture was developed, every program anyone wanted to run on it had to be rewritten almost from scratch. Because of this, computer architects tried hard to design systems that were backward-compatible with their previous systems, so that the new and improved models could run the same programs as the previous models. For example, the current generation of PC-clones are compatible with their 1982 ancestors, and current IBM 390-series machines will run the same software as the legendary IBM mainframes of the 1960’s.

To make matters worse, programming in assembly language is time-consuming and difficult. Early software engineering studies indicated that programmers wrote about as many lines of code per year no matter what language they used. Therefore, a programmer who used a high-level language, in which a single line of code was equivalent to five lines of assembly language code, could be about five times more productive than a programmer working in assembly language. It’s not surprising, therefore, that a great deal of energy has been devoted to developing high-level languages where a single statement might represent dozens of lines of assembly language, and will run without modification on many different computers.

By the mid-1980s, the following trends had become apparent:

- Few people were doing assembly language programming any longer if they could possibly
avoid it.

- Compilers for high-level languages only used a fraction of the instructions available in the
assembly languages of the more complex architectures.

- Computer architects were discovering new ways to make computers faster, using techniques
that would be difficult to implement in existing architectures.

At various times, experimental computer architectures that took advantage of these trends were
developed. The lessons learned from these architectures eventually evolved into the \textit{RISC} (Re-
duced Instruction Set Computer) philosophy.

The exact definition of RISC is difficult to state\footnote{It seems to be an axiom of Computer Science that for every known definition of RISC, there exists someone who strongly disagrees with it.}, but the basic characteristic of a RISC archi-
tecture, from the point of view of an assembly language programmer, is that the instruction set is
relatively small and simple compared to the instruction sets of more traditional architectures (now
often referred to as \textit{CISC}, or Complex Instruction Set Computers).

The MIPS architecture is one example of a RISC architecture, but there are many others.

\section{MIPS Instruction Set Overview}

In this and the following sections we will give details of the MIPS architecture and SPIM envi-
ronment sufficient for many purposes. Readers who want even more detail should consult \textit{SPIM
S20: A MIPS R2000 Simulator} by James Larus, \textit{Appendix A, Computer Organization and Design}
by David Patterson and John Hennessy (this appendix is an expansion of the SPIM S20 document
by James Larus), or \textit{MIPS R2000 RISC Architecture} by Gerry Kane.

The MIPS architecture is a register architecture. All arithmetic and logical operations involve
only registers (or constants that are stored as part of the instructions). The MIPS architecture also
includes several simple instructions for loading data from memory into registers and storing data
from registers in memory; for this reason, the MIPS architecture is called a \textit{load/store} architecture.
In a load/store (or \textit{load and store}) architecture, the only instructions that can access memory are
the \textit{load} and \textit{store} instructions— all other instructions access only registers.

\section{The MIPS Register Set}

The MIPS R2000 CPU has 32 registers. 31 of these are general-purpose registers that can be used
in any of the instructions. The last one, denoted register \texttt{zero}, is defined to contain the number
zero at all times.
Even though any of the registers can theoretically be used for any purpose, MIPS programmers have agreed upon a set of guidelines that specify how each of the registers should be used. Programmers (and compilers) know that as long as they follow these guidelines, their code will work properly with other MIPS code.

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>Constant 0.</td>
</tr>
<tr>
<td>at</td>
<td>1</td>
<td>Reserved for the assembler.</td>
</tr>
<tr>
<td>v0 - v1</td>
<td>2 - 3</td>
<td>Result Registers.</td>
</tr>
<tr>
<td>a0 - a3</td>
<td>4 - 7</td>
<td>Argument Registers 1 - 4.</td>
</tr>
<tr>
<td>t0 - t9</td>
<td>8 - 15, 24 - 25</td>
<td>Temporary Registers 0 - 9.</td>
</tr>
<tr>
<td>s0 - s7</td>
<td>16 - 23</td>
<td>Saved Registers 0 - 7.</td>
</tr>
<tr>
<td>k0 - k1</td>
<td>26 - 27</td>
<td>Kernel Registers 0 - 1.</td>
</tr>
<tr>
<td>gp</td>
<td>28</td>
<td>Global Data Pointer.</td>
</tr>
<tr>
<td>sp</td>
<td>29</td>
<td>Stack Pointer.</td>
</tr>
<tr>
<td>fp</td>
<td>30</td>
<td>Frame Pointer.</td>
</tr>
<tr>
<td>ra</td>
<td>31</td>
<td>Return Address.</td>
</tr>
</tbody>
</table>

8.4 The MIPS Instruction Set

This section briefly describes the MIPS assembly language instruction set.

In the description of the instructions, the following notation is used:

- If an instruction description begins with an `o`, then the instruction is not a member of the native MIPS instruction set, but is available as a *pseudoinstruction*. The assembler translates pseudoinstructions into one or more native instructions (see section 8.7 and exercise 8.8.1 for more information).

- If the op contains a `(u)`, then this instruction can either use signed or unsigned arithmetic, depending on whether or not a `u` is appended to the name of the instruction. For example, if the op is given as `add(u)`, then this instruction can either be `add` (add signed) or `addu` (add unsigned).

- `des` must always be a register.

- `src1` must always be a register.

- `reg2` must always be a register.

- `src2` may be either a register or a 32-bit integer.
- `addr` must be an address. See section 8.4.4 for a description of valid addresses.

### 8.4.1 Arithmetic Instructions

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>des, src1</td>
<td>des gets the absolute value of src1.</td>
</tr>
<tr>
<td>add (u)</td>
<td>des, src1, src2</td>
<td>des gets src1 + src2.</td>
</tr>
<tr>
<td>and</td>
<td>des, src1, src2</td>
<td>des gets the bitwise and of src1 and src2.</td>
</tr>
<tr>
<td>div (u)</td>
<td>src1, reg2</td>
<td>Divide src1 by reg2, leaving the quotient in register lo and the remainder in register hi.</td>
</tr>
<tr>
<td>div (u)</td>
<td>des, src1, src2</td>
<td>des gets src1 / src2.</td>
</tr>
<tr>
<td>mul</td>
<td>des, src1, src2</td>
<td>des gets src1 × src2.</td>
</tr>
<tr>
<td>mulo</td>
<td>des, src1, src2</td>
<td>des gets src1 × src2, with overflow.</td>
</tr>
<tr>
<td>mult (u)</td>
<td>src1, reg2</td>
<td>Multiply src1 and reg2, leaving the low-order word in register lo and the high-order word in register hi.</td>
</tr>
<tr>
<td>neg (u)</td>
<td>des, src1</td>
<td>des gets the negative of src1.</td>
</tr>
<tr>
<td>nor</td>
<td>des, src1, src2</td>
<td>des gets the bitwise logical nor of src1 and src2.</td>
</tr>
<tr>
<td>not</td>
<td>des, src1</td>
<td>des gets the bitwise logical negation of src1.</td>
</tr>
<tr>
<td>or</td>
<td>des, src1, src2</td>
<td>des gets the bitwise logical or of src1 and src2.</td>
</tr>
<tr>
<td>rem (u)</td>
<td>des, src1, src2</td>
<td>des gets the remainder of dividing src1 by src2.</td>
</tr>
<tr>
<td>rol</td>
<td>des, src1, src2</td>
<td>des gets the result of rotating left the contents of src1 by src2 bits.</td>
</tr>
<tr>
<td>ror</td>
<td>des, src1, src2</td>
<td>des gets the result of rotating right the contents of src1 by src2 bits.</td>
</tr>
<tr>
<td>sll</td>
<td>des, src1, src2</td>
<td>des gets src1 shifted left by src2 bits.</td>
</tr>
<tr>
<td>sra</td>
<td>des, src1, src2</td>
<td>Right shift arithmetic.</td>
</tr>
<tr>
<td>srl</td>
<td>des, src1, src2</td>
<td>Right shift logical.</td>
</tr>
<tr>
<td>sub (u)</td>
<td>des, src1, src2</td>
<td>des gets src1 - src2.</td>
</tr>
<tr>
<td>xor</td>
<td>des, src1, src2</td>
<td>des gets the bitwise exclusive or of src1 and src2.</td>
</tr>
</tbody>
</table>
8.4. THE MIPS INSTRUCTION SET

8.4.2 Comparison Instructions

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>des, src1, src2</td>
<td>( \text{des} \leftarrow 1 ) if ( \text{src1} = \text{src2} ), 0 otherwise.</td>
</tr>
<tr>
<td>sne</td>
<td>des, src1, src2</td>
<td>( \text{des} \leftarrow 1 ) if ( \text{src1} \neq \text{src2} ), 0 otherwise.</td>
</tr>
<tr>
<td>sge(u)</td>
<td>des, src1, src2</td>
<td>( \text{des} \leftarrow 1 ) if ( \text{src1} \geq \text{src2} ), 0 otherwise.</td>
</tr>
<tr>
<td>sgt(u)</td>
<td>des, src1, src2</td>
<td>( \text{des} \leftarrow 1 ) if ( \text{src1} &gt; \text{src2} ), 0 otherwise.</td>
</tr>
<tr>
<td>sle(u)</td>
<td>des, src1, src2</td>
<td>( \text{des} \leftarrow 1 ) if ( \text{src1} \leq \text{src2} ), 0 otherwise.</td>
</tr>
<tr>
<td>slt(u)</td>
<td>des, src1, src2</td>
<td>( \text{des} \leftarrow 1 ) if ( \text{src1} &lt; \text{src2} ), 0 otherwise.</td>
</tr>
</tbody>
</table>

8.4.3 Branch and Jump Instructions

8.4.3.1 Branch

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>lab</td>
<td>Unconditional branch to ( \text{lab} ).</td>
</tr>
<tr>
<td>beq</td>
<td>src1, src2, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \equiv \text{src2} ).</td>
</tr>
<tr>
<td>bne</td>
<td>src1, src2, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \neq \text{src2} ).</td>
</tr>
<tr>
<td>bge(u)</td>
<td>src1, src2, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \geq \text{src2} ).</td>
</tr>
<tr>
<td>bgt(u)</td>
<td>src1, src2, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} &gt; \text{src2} ).</td>
</tr>
<tr>
<td>ble(u)</td>
<td>src1, src2, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \leq \text{src2} ).</td>
</tr>
<tr>
<td>blt(u)</td>
<td>src1, src2, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} &lt; \text{src2} ).</td>
</tr>
<tr>
<td>beqz</td>
<td>src1, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \equiv 0 ).</td>
</tr>
<tr>
<td>bnez</td>
<td>src1, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \neq 0 ).</td>
</tr>
<tr>
<td>bgez</td>
<td>src1, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \geq 0 ).</td>
</tr>
<tr>
<td>bgtz</td>
<td>src1, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} &gt; 0 ).</td>
</tr>
<tr>
<td>blez</td>
<td>src1, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} \leq 0 ).</td>
</tr>
<tr>
<td>bltz</td>
<td>src1, lab</td>
<td>Branch to ( \text{lab} ) if ( \text{src1} &lt; 0 ).</td>
</tr>
<tr>
<td>bgezal</td>
<td>src1, lab</td>
<td>If ( \text{src1} \geq 0 ), then put the address of the next instruction into ( \text{$ra} ) and branch to ( \text{lab} ).</td>
</tr>
<tr>
<td>bgtzal</td>
<td>src1, lab</td>
<td>If ( \text{src1} &gt; 0 ), then put the address of the next instruction into ( \text{$ra} ) and branch to ( \text{lab} ).</td>
</tr>
<tr>
<td>bltzal</td>
<td>src1, lab</td>
<td>If ( \text{src1} &lt; 0 ), then put the address of the next instruction into ( \text{$ra} ) and branch to ( \text{lab} ).</td>
</tr>
</tbody>
</table>
8.4.3.2 Jump

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>label</td>
<td>Jump to label lab.</td>
</tr>
<tr>
<td>jr</td>
<td>src1</td>
<td>Jump to location src1.</td>
</tr>
<tr>
<td>jal</td>
<td>label</td>
<td>Jump to label lab, and store the address of the next instruction in $ra.</td>
</tr>
<tr>
<td>jalr</td>
<td>src1</td>
<td>Jump to location src1, and store the address of the next instruction in $ra.</td>
</tr>
</tbody>
</table>

8.4.4 Load, Store, and Data Movement

The second operand of all of the load and store instructions must be an address. The MIPS architecture supports the following addressing modes:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reg)</td>
<td>Contents of reg.</td>
</tr>
<tr>
<td>const</td>
<td>A constant address.</td>
</tr>
<tr>
<td>const(reg)</td>
<td>const + contents of reg.</td>
</tr>
<tr>
<td>symbol</td>
<td>The address of symbol.</td>
</tr>
<tr>
<td>symbol+const</td>
<td>The address of symbol + const.</td>
</tr>
<tr>
<td>symbol+const(reg)</td>
<td>The address of symbol + const + contents of reg.</td>
</tr>
</tbody>
</table>

8.4.4.1 Load

The load instructions, with the exceptions of li and lui, fetch a byte, halfword, or word from memory and put it into a register. The li and lui instructions load a constant into a register.

All load addresses must be aligned on the size of the item being loaded. For example, all loads of halfwords must be from even addresses, and loads of words from addresses cleanly divisible by four. The ulh and ulw instructions are provided to load halfwords and words from addresses that might not be aligned properly.
### 8.4. THE MIPS INSTRUCTION SET

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>la</td>
<td>des, addr</td>
<td>Load the address of a label.</td>
</tr>
<tr>
<td>lb(u)</td>
<td>des, addr</td>
<td>Load the byte at addr into des.</td>
</tr>
<tr>
<td>lh(u)</td>
<td>des, addr</td>
<td>Load the halfword at addr into des.</td>
</tr>
<tr>
<td>li</td>
<td>des, const</td>
<td>Load the constant const into des.</td>
</tr>
<tr>
<td>lui</td>
<td>des, const</td>
<td>Load the constant const into the upper halfword of des, and set the lower halfword of des to 0.</td>
</tr>
<tr>
<td>lw</td>
<td>des, addr</td>
<td>Load the word at addr into des.</td>
</tr>
<tr>
<td>lwl</td>
<td>des, addr</td>
<td>Store the lower byte of register src1 to addr.</td>
</tr>
<tr>
<td>lwr</td>
<td>des, addr</td>
<td>Store the lower halfword of register src1 to addr.</td>
</tr>
<tr>
<td>ulh(u)</td>
<td>des, addr</td>
<td>Load the halfword starting at the (possibly unaligned) address addr into des.</td>
</tr>
<tr>
<td>ulw</td>
<td>des, addr</td>
<td>Load the word starting at the (possibly unaligned) address addr into des.</td>
</tr>
</tbody>
</table>

#### 8.4.4.2 Store

The store instructions store a byte, halfword, or word from a register into memory.

Like the load instructions, all store addresses must be aligned on the size of the item being stored. For example, all stores of halfwords must be from even addresses, and loads of words from addresses cleanly divisible by four. The swl, swr, ush and usw instructions are provided to store halfwords and words to addresses which might not be aligned properly.

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sb</td>
<td>src1, addr</td>
<td>Store the lower byte of register src1 to addr.</td>
</tr>
<tr>
<td>sh</td>
<td>src1, addr</td>
<td>Store the lower halfword of register src1 to addr.</td>
</tr>
<tr>
<td>sw</td>
<td>src1, addr</td>
<td>Store the word in register src1 to addr.</td>
</tr>
<tr>
<td>swl</td>
<td>src1, addr</td>
<td>Store the upper halfword in src to the (possibly unaligned) address addr.</td>
</tr>
<tr>
<td>swr</td>
<td>src1, addr</td>
<td>Store the lower halfword in src to the (possibly unaligned) address addr.</td>
</tr>
<tr>
<td>ush</td>
<td>src1, addr</td>
<td>Store the lower halfword in src to the (possibly unaligned) address addr.</td>
</tr>
<tr>
<td>usw</td>
<td>src1, addr</td>
<td>Store the word in src to the (possibly unaligned) address addr.</td>
</tr>
</tbody>
</table>
8.4.4.3 Data Movement

The data movement instructions move data among registers. Special instructions are provided to move data in and out of special registers such as \texttt{hi} and \texttt{lo}.

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>des, src1</td>
<td>Copy the contents of src1 to des.</td>
</tr>
<tr>
<td>mfhi</td>
<td>des</td>
<td>Copy the contents of the hi register to des.</td>
</tr>
<tr>
<td>mflo</td>
<td>des</td>
<td>Copy the contents of the lo register to des.</td>
</tr>
<tr>
<td>mthi</td>
<td>src1</td>
<td>Copy the contents of the src1 to hi.</td>
</tr>
<tr>
<td>mtlo</td>
<td>src1</td>
<td>Copy the contents of the src1 to lo.</td>
</tr>
</tbody>
</table>

8.4.5 Exception Handling

<table>
<thead>
<tr>
<th>Op</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfe</td>
<td></td>
<td>Return from exception.</td>
</tr>
<tr>
<td>syscall</td>
<td></td>
<td>Makes a system call. See 8.6.1 for a list of the SPIM system calls.</td>
</tr>
<tr>
<td>break</td>
<td>const</td>
<td>Used by the debugger.</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td>An instruction which has no effect (other than taking a cycle to execute).</td>
</tr>
</tbody>
</table>
8.5 The SPIM Assembler

8.5.1 Segment and Linker Directives

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.data</td>
<td>addr</td>
<td>The following items are to be assembled into the data segment. By default, begin at the next available address in the data segment. If the optional argument <code>addr</code> is present, then begin at <code>addr</code>.</td>
</tr>
<tr>
<td>.text</td>
<td>addr</td>
<td>The following items are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument <code>addr</code> is present, then begin at <code>addr</code>. In SPIM, the only items that can be assembled into the text segment are instructions and words (via the <code>.word</code> directive).</td>
</tr>
<tr>
<td>.kdata</td>
<td>addr</td>
<td>The kernel data segment. Like the data segment, but used by the Operating System.</td>
</tr>
<tr>
<td>.ktext</td>
<td>addr</td>
<td>The kernel text segment. Like the text segment, but used by the Operating System.</td>
</tr>
<tr>
<td>.extern</td>
<td>sym size</td>
<td>Declare as global the label <code>sym</code>, and declare that it is <code>size</code> bytes in length (this information can be used by the assembler).</td>
</tr>
<tr>
<td>.globl</td>
<td>sym</td>
<td>Declare as global the label <code>sym</code>.</td>
</tr>
</tbody>
</table>
8.5.2 Data Directives

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.align</td>
<td>n</td>
<td>Align the next item on the next (2^n)-byte boundary. (.align 0) turns off automatic alignment.</td>
</tr>
<tr>
<td>.ascii</td>
<td>str</td>
<td>Assemble the given string in memory. Do not null-terminate.</td>
</tr>
<tr>
<td>.asciiz</td>
<td>str</td>
<td>Assemble the given string in memory. Do null-terminate.</td>
</tr>
<tr>
<td>.byte</td>
<td>byte1 \ldots byteN</td>
<td>Assemble the given bytes (8-bit integers).</td>
</tr>
<tr>
<td>.half</td>
<td>half1 \ldots halfN</td>
<td>Assemble the given halfwords (16-bit integers).</td>
</tr>
<tr>
<td>.space</td>
<td>size</td>
<td>Allocate (n) bytes of space in the current segment. In SPIM, this is only permitted in the data segment.</td>
</tr>
<tr>
<td>.word</td>
<td>word1 \ldots wordN</td>
<td>Assemble the given words (32-bit integers).</td>
</tr>
</tbody>
</table>

8.6 The SPIM Environment

8.6.1 SPIM syscalls

<table>
<thead>
<tr>
<th>Service</th>
<th>Code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0</td>
<td>none</td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12</td>
<td>none</td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12</td>
<td>none</td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0</td>
<td>none</td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td>none</td>
<td>$v0</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td>none</td>
<td>$f0</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td>none</td>
<td>$f0</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 (address), $a1 (length)</td>
<td>none</td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 (length)</td>
<td>$v0</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

8.7 The Native MIPS Instruction Set

Many of the instructions listed here are not native MIPS instructions. Instead, they are pseudo-instructions—macros that the assembler knows how to translate into native MIPS instructions. Instead of programming the “real” hardware, MIPS programmers generally use the virtual machine
implemented by the MIPS assembler, which is much easier to program than the native machine.

For example, in most cases, the SPIM assembler will allow src2 to be a 32-bit integer constant. Of course, since the MIPS instructions are all exactly 32 bits in length, there’s no way that a 32-bit constant can fit in a 32-bit instruction word and have any room left over to specify the operation and the operand registers! When confronted with a 32-bit constant, the assembler uses a table of rules to generate a sequence of native instructions that will do what the programmer has asked.

The assembler also performs some more intricate transformations to translate your programs into a sequence of native MIPS instructions, but these will not be discussed in this text.

By default, the SPIM environment implements the same virtual machine that the MIPS assembler uses. It also implements the bare machine, if invoked with the \texttt{-bare} option enabled.
8.8 Exercises

8.8.1

Many of the instructions available to the MIPS assembly language programmer are not really instructions at all, but are translated by the assembler into one or more instructions.

For example, the move instruction can be implemented using the add instruction. Making use of register $0, which always contains the constant zero, and the fact that the for any number $x$, $x + 0 \equiv x$, we can rewrite

\[
\text{move des, src1}
\]

as

\[
\text{add des, src1, $0$}
\]

Similarly, since either the exclusive or or inclusive or of any number and 0 gives the number, we could also write this as either of the following:

\[
\text{or des, src1, $0$}
\]
\[
\text{xor des, src1, $0$}
\]

Show how you could implement the following instructions, using other instructions in the native MIPS instruction set:

1. \text{rem des, src1, src2}
2. \text{mul des, src1, src2}
3. \text{li des, const}
4. \text{lui des, const}

Keep in mind that the register $at$ is reserved for use by the assembler, so you can feel free to use this register for scratch space. You must not clobber any other registers, however.
Chapter 9

MIPS Assembly Code Examples

by Daniel J. Ellard

The following sections include the source code for several of the programs referenced by the tutorial. All of this source code is also available online.

For the convenience of the reader, the source code is listed here along with line numbers in the left margin. These line numbers do not appear in the original code, and it would be an error to include them in your own code.
9.1 add2.asm

This program is described in section 7.4.

```
1 ## Daniel J. Ellard -- 02/21/94
2 ## add2.asm-- A program that computes and prints the sum
3 ## of two numbers specified at runtime by the user.
4 ## Registers used:
5 ## $t0 - used to hold the first number.
6 ## $t1 - used to hold the second number.
7 ## $t2 - used to hold the sum of the $t1 and $t2.
8 ## $v0 - syscall parameter and return value.
9 ## $a0 - syscall parameter.
10
11 main:
12     ## Get first number from user, put into $t0.
13     li    $v0, 5   # load syscall read_int into $v0.
14     syscall # make the syscall.
15     move   $t0, $v0 # move the number read into $t0.
16
17     ## Get second number from user, put into $t1.
18     li    $v0, 5   # load syscall read_int into $v0.
19     syscall # make the syscall.
20     move   $t1, $v0 # move the number read into $t1.
21
22     add    $t2, $t0, $t1 # compute the sum.
23
24     ## Print out $t2.
25     move   $a0, $t2 # move the number to print into $a0.
26     li    $v0, 1   # load syscall print_int into $v0.
27     syscall # make the syscall.
28
29     li    $v0, 10 # syscall code 10 is for exit.
30     syscall # make the syscall.
31
32 ## end of add2.asm.
```
This program is described in section 7.5.

```
1 ## Daniel J. Ellard -- 02/21/94
2 ## hello.asm-- A "Hello World" program.
3 ## Registers used:
4 ## $v0   - syscall parameter and return value.
5 ## $a0   - syscall parameter-- the string to print.
6
7 .text
8 main:
9 la     $a0, hello_msg  # load the addr of hello_msg into $a0.
10 li     $v0, 4         # 4 is the print_string syscall.
11 syscall # do the syscall.
12
13 li     $v0, 10        # 10 is the exit syscall.
14 syscall # do the syscall.
15
16 ## Data for the program:
17 .data
18 hello_msg:  .asciiz "Hello World\n"
19
20 ## end hello.asm
```
9.3 multiples.asm

This program is described in section 7.7. The algorithm used is algorithm 7.1 (shown on page 230).

```assembly
1 ## Daniel J. Ellard -- 02/21/94
2 ## multiples.asm-- takes two numbers A and B, and prints out
3 ## all the multiples of A from A to A * B.
4 ## If B <= 0, then no multiples are printed.
5 ## Registers used:
6 ## $t0 - used to hold A.
7 ## $t1 - used to hold B.
8 ## $t2 - used to store S, the sentinel value A * B.
9 ## $t3 - used to store m, the current multiple of A.
10
11 .text
12 main:
13 ## read A into $t0, B into $t1.
14 li $v0, 5 # syscall 5 = read_int
15 syscall
16 move $t0, $v0 # A = integer just read
17
18 li $v0, 5 # syscall 5 = read_int
19 syscall
20 move $t1, $v0 # B = integer just read
21
22 blez $t1, exit # if B <= 0, exit.
23
24 mul $t2, $t0, $t1 # S = A * B.
25 move $t3, $t0 # m = A
26
27 loop:
28 move $a0, $t3 # print m.
29 li $v0, 1 # syscall 1 = print_int
30 syscall # make the system call.
31
32 beq $t2, $t3, endloop # if m == S, we’re done.
33 add $t3, $t3, $t0 # otherwise, m = m + A.
34
35 la $a0, space # print a space.
36 li $v0, 4 # syscall 4 = print_string
37 syscall
38
39 b loop # iterate.
40 endloop:
```
```assembly
41    la    $a0, newline            # print a newline:
42    li    $v0, 4                 # syscall 4 = print_string
43    syscall
44
45    exit:                        # exit the program:
46    li    $v0, 10                # syscall 10 = exit
47    syscall                     # we're outta here.
48
49    ## Here's where the data for this program is stored:
50    .data
51    space:                       .asciiz " 
52    newline:                     .asciiz "\n"
53
54    ## end of multiples.asm
```
9.4 palindrome.asm

This program is described in section 7.8. The algorithm used is algorithm 7.2 (shown on page 232).

```assembly
1 ## Daniel J. Ellard -- 02/21/94
2 ## palindrome.asm -- read a line of text and test if it is a palindrome.
3 ## Register usage:
4 ## $t1 - A.
5 ## $t2 - B.
6 ## $t3 - the character at address A.
7 ## $t4 - the character at address B.
8 ## $v0 - syscall parameter / return values.
9 ## $a0 - syscall parameters.
10 ## $a1 - syscall parameters.

11 .text
12 main: # SPIM starts by jumping to main.
13 ## read the string S:
14 la $a0, string_space  # read the string S:
15 li $a1, 1024
16 li $v0, 8  # load "read_string" code into $v0.
17 syscall
18
19 la $t1, string_space # A = S.
20
21 la $t2, string_space ## we need to move B to the end
22 length_loop: # of the string:
23 lb $t3, ($t2) # load the byte at addr B into $t3.
24 beqz $t3, end_length_loop # if $t3 == 0, branch out of loop.
25 addu $t2, $t2, 1 # otherwise, increment B,
26 b length_loop # and repeat the loop.
27 end_length_loop:
28 subu $t2, $t2, 2 ## subtract 2 to move B back past
29 # the '" and '\n'.
30
31 test_loop:
32 bge $t1, $t2, is_palin # if A >= B, it's a palindrome.
33
34 lb $t3, ($t1) # load the byte at addr A into $t3,
35 lb $t4, ($t2) # load the byte at addr B into $t4.
36 bne $t3, $t4, not_palin # if $t3 != $t4, not a palindrome.
37 b not_palin
38 addu $t1, $t1, 1 # increment A,
39 subu $t2, $t2, 1 # decrement B,
40 b test_loop # and repeat the loop.
```

266  CHAPTER 9.  MIPS ASSEMBLY CODE EXAMPLES
9.4. PALINDROME.ASM

41
42 is_palin:                   ## print the is_palin_msg, and exit.
43    la $a0, is_palin_msg
44    li $v0, 4
45    syscall
46    b exit
47
48 not_palin:
49    la $a0, not_palin_msg    ## print the is_palin_msg, and exit.
50    li $v0, 4
51    syscall
52    b exit
53
54 exit:                        ## exit the program:
55    li $v0, 10
56    syscall
57
58 ## Here’s where the data for this program is stored:
59 .data
60 string_space: .space 1024    # reserve 1024 bytes for the string.
61 is_palin_msg: .asciiz "The string is a palindrome.\n"
62 not_palin_msg: .asciiz "The string is not a palindrome.\n"
63
64 ## end of palindrome.asm
This program is described in section 7.9.1. The algorithm used is algorithm 7.3 (shown on page 235).

```assembly
    ## Daniel J. Ellard -- 03/02/94
    ## atoi-1.asm -- reads a line of text, converts it to an integer, and
    ## prints the integer.
    ## Register usage:
    ## $t0  - S.
    ## $t1  - the character pointed to by S.
    ## $t2  - the current sum.

    .text
    main:
        la  $a0, string_space ## read the string S:
        li  $a1, 1024
        li  $v0, 8 # load "read_string" code into $v0.
        syscall

        la  $t0, string_space # Initialize S.
        li  $t2, 0 # Initialize sum = 0.

        sum_loop:
        lb  $t1, ($t0) # load the byte at addr S into $t1,
        addu $t0, $t0, 1 # and increment S.

        ## use 10 instead of \n due to SPIM bug!
        beq  $t1, 10, end_sum_loop # if $t1 == \n, branch out of loop.

        mul  $t2, $t2, 10 # t2 *= 10.
        sub  $t1, $t1, '0' # t1 -= '0'.
        add  $t2, $t2, $t1 # t2 += t1.

        b    sum_loop # and repeat the loop.

    end_sum_loop:
    move  $a0, $t2 # print out the answer (t2).
    li    $v0, 1
    syscall

    la    $a0, newline # and then print out a newline.
    li    $v0, 4
    syscall
```
exit:
li $v0, 10
syscall

.data
newline: .asciiz "\n"
string_space: .space 1024

## end of atoi-1.asm
### 9.6 atoi-4.asm

This program is described in section 7.9.4. The algorithm used is algorithm 7.3 (shown on page 235), modified as described in section 7.9.4.

```assembly
## Daniel J. Ellard -- 03/04/94
## atoi-4.asm -- reads a line of text, converts it to an integer, and prints the integer.
## Handles signed numbers, detects bad characters, and overflow.
## Register usage:
## $t0  - S.
## $t1  - the character pointed to by S.
## $t2  - the current sum.
## $t3  - the "sign" of the sum.
## $t4  - holds the constant 10.
## $t5  - used to test for overflow.
.text
main:
    la $a0, string_space # read the string S:
    li $a1, 1024
    li $v0, 8 # load "read_string" code into $v0.
    syscall

    la $t0, string_space # Initialize S.
    li $t2, 0 # Initialize sum = 0.

get_sign:
    li $t3, 1 # assume the sign is positive.
    lb $t1, ($t0) # grab the "sign"
    bne $t1, '-', positive # if not "-", do nothing.
    li $t3, -1 # otherwise, set t3 = -1, and
    addu $t0, $t0, 1 # skip over the sign.
positive:
    li $t4, 10 # store the constant 10 in $t4.
sum_loop:
    lb $t1, ($t0) # load the byte at addr S into $t1,
    addu $t0, $t0, 1 # and increment S,
    mult $t2, $t4 # multiply $t2 by 10.
```

---

## 9.6.1 atoi-4.asm

This program is described in section 7.9.4. The algorithm used is algorithm 7.3 (shown on page 235), modified as described in section 7.9.4.

```assembly
## Daniel J. Ellard -- 03/04/94
## atoi-4.asm -- reads a line of text, converts it to an integer, and prints the integer.
## Handles signed numbers, detects bad characters, and overflow.
## Register usage:
## $t0  - S.
## $t1  - the character pointed to by S.
## $t2  - the current sum.
## $t3  - the "sign" of the sum.
## $t4  - holds the constant 10.
## $t5  - used to test for overflow.
.text
main:
    la $a0, string_space # read the string S:
    li $a1, 1024
    li $v0, 8 # load "read_string" code into $v0.
    syscall

    la $t0, string_space # Initialize S.
    li $t2, 0 # Initialize sum = 0.

get_sign:
    li $t3, 1 # assume the sign is positive.
    lb $t1, ($t0) # grab the "sign"
    bne $t1, '-', positive # if not "-", do nothing.
    li $t3, -1 # otherwise, set t3 = -1, and
    addu $t0, $t0, 1 # skip over the sign.
positive:
    li $t4, 10 # store the constant 10 in $t4.
sum_loop:
    lb $t1, ($t0) # load the byte at addr S into $t1,
    addu $t0, $t0, 1 # and increment S,
    mult $t2, $t4 # multiply $t2 by 10.
```

---

## 9.6.1 atoi-4.asm

This program is described in section 7.9.4. The algorithm used is algorithm 7.3 (shown on page 235), modified as described in section 7.9.4.

```assembly
## Daniel J. Ellard -- 03/04/94
## atoi-4.asm -- reads a line of text, converts it to an integer, and prints the integer.
## Handles signed numbers, detects bad characters, and overflow.
## Register usage:
## $t0  - S.
## $t1  - the character pointed to by S.
## $t2  - the current sum.
## $t3  - the "sign" of the sum.
## $t4  - holds the constant 10.
## $t5  - used to test for overflow.
.text
main:
    la $a0, string_space # read the string S:
    li $a1, 1024
    li $v0, 8 # load "read_string" code into $v0.
    syscall

    la $t0, string_space # Initialize S.
    li $t2, 0 # Initialize sum = 0.

get_sign:
    li $t3, 1 # assume the sign is positive.
    lb $t1, ($t0) # grab the "sign"
    bne $t1, '-', positive # if not "-", do nothing.
    li $t3, -1 # otherwise, set t3 = -1, and
    addu $t0, $t0, 1 # skip over the sign.
positive:
    li $t4, 10 # store the constant 10 in $t4.
sum_loop:
    lb $t1, ($t0) # load the byte at addr S into $t1,
    addu $t0, $t0, 1 # and increment S,
    mult $t2, $t4 # multiply $t2 by 10.
```
9.6. ATOI-4.ASM

41      mfhi $t5                       # check for overflow;
42      bnez $t5, overflow           # if so, then report an overflow.
43      mflo $t2                      # get the result of the multiply
44      blt $t2, $0, overflow        # make sure that it isn’t negative.
45
46      sub $t1, $t1, ’0’           # t1 -= ’0’.
47      add $t2, $t2, $t1           # t2 += t1.
48      blt $t2, $0, overflow       # and repeat the loop.
49
50      b sum_loop                  # and repeat the loop.
51      end_sum_loop:
52          mul $t2, $t2, $t3        # set the sign properly.
53
54      move $a0, $t2               # print out the answer (t2).
55      li $v0, 1
56      syscall
57
58      la $a0, newline            # and then print out a newline.
59      li $v0, 4
60      syscall
61
62      b exit                     # indicate that an overflow occurred.
63
64      overflow:
65          la $a0, overflow_msg     # exit the program:
66          li $v0, 4
67          syscall
68          b exit
69
70      exit:
71          li $v0, 10              # print out the answer (t2).
72          syscall
73
74      .data
75      newline: .asciiz "\n"
76      overflow_msg: .asciiz "Overflow!\n"
77      string_space: .space 1024  # reserve 1024 bytes for the string.
78
79      ## end of atoi-4.asm
CHAPTER 9. MIPS ASSEMBLY CODE EXAMPLES

9.7 printf.asm

Using syscalls for output can quickly become tedious, and output routines can quickly muddy up even the neatest code, since it requires several assembly instructions just to print out a number. To make matters worse, there is no syscall which prints out a single ASCII character.

To help my own coding, I wrote the following \texttt{printf} function, which behaves like a simplified form of the \texttt{printf} function in the standard C library. It implements only a fraction of the functionality of the real \texttt{printf}, but enough to be useful. See the comments in the code for more information.


daniel.j.ellard -- 03/13/94

\texttt{printf.asm--}

an implementation of a simple printf work-alike.

\texttt{printf--}

A simple printf-like function. Understands just the basic forms of the \%s, \%d, \%c, and \%\% formats, and can only have 3 embedded formats (so that all of the parameters are passed in registers).

If there are more than 3 embedded formats, all but the first 3 are completely ignored (not even printed).

Register Usage:

\texttt{\$a0,\$s0} - pointer to format string
\texttt{\$a1,\$s1} - format argument 1 (optional)
\texttt{\$a2,\$s2} - format argument 2 (optional)
\texttt{\$a3,\$s3} - format argument 3 (optional)
\texttt{\$s4} - count of formats processed.
\texttt{\$s5} - char at \$s4.
\texttt{\$s6} - pointer to printf buffer

.text
.globl printf
.printf:

\texttt{subu \$sp, \$sp, 36} # set up the stack frame,
\texttt{sw \$ra, 32(\$sp)} # saving the local environment.
\texttt{sw \$fp, 28(\$sp)}
\texttt{sw \$s0, 24(\$sp)}
\texttt{sw \$s1, 20(\$sp)}
\texttt{sw \$s2, 16(\$sp)}
\texttt{sw \$s3, 12(\$sp)}
\texttt{sw \$s4, 8(\$sp)}
\texttt{sw \$s5, 4(\$sp)}
\texttt{sw \$s6, 0(\$sp)}
\texttt{addu \$fp, \$sp, 36}
35  move $s0, $a0
# grab the arguments:
36  move $s1, $a1
# fmt string
37  move $s2, $a2
# arg1 (optional)
38  move $s3, $a3
# arg2 (optional)
39  li $s4, 0
# set # of formats = 0
40  la $s6, printf_buf
# set s6 = base of printf buffer.
41  printf_loop:
# process each character in the fmt:
42    lb $s5, 0($s0)
# get the next character, and then
43    addu $s0, $s0, 1
# bump up $s0 to the next character.
44    beq $s5, '%', printf_fmt
# if the fmt character, then do fmt.
45    beq $0, $s5, printf_end
# if zero, then go to end.
46    printf_putc:
51  printf_putc:
52    sb $s5, 0($s6)
# otherwise, just put this char
53    sb $0, 1($s6)
# into the printf buffer,
54    move $a0, $s6
# and then print it with the
55    li $v0, 4
# print_str syscall
56    syscall
57    b printf_loop
# loop on.
58  printf_fmt:
59    lb $s5, 0($s0)
# see what the fmt character is,
60    addu $s0, $s0, 1
# and bump up the pointer.
61    beq $s4, 3, printf_loop
# if we've already processed 3 args,
62    beq $s5, 'd', printf_int
# then *ignore* this fmt.
63    beq $s5, 's', printf_str
# if 'd', print as a decimal integer.
64    beq $s5, 'c', printf_char
# if 's', print as a string.
65    beq $s5, '%', printf_perc
# if 'c', print as an ASCII char.
66    beq $s5, '%', printf_loop
# if '%', print a '%'
67    b printf_shift_args
# otherwise, just continue.
68  printf_shift_args:
69    move $s1, $s2
# shift over the fmt args,
70    move $s2, $s3
# $s1 = $s2
71    add $s4, $s4, 1
# $s2 = $s3
72    move $s3, $a3
# increment # of args processed.
73    b printf_loop
# and continue the main loop.
CHAPTER 9. MIPS ASSEMBLY CODE EXAMPLES

80  printf_int: # deal with a %d:
81       move $a0, $s1 # do a print_int syscall of $s1.
82       li $v0, 1
83       syscall
84       b printf_shift_args # branch to printf_shift_args
85
86  printf_str: # deal with a %s:
87       move $a0, $s1 # do a print_string syscall of $s1.
88       li $v0, 4
89       syscall
90       b printf_shift_args # branch to printf_shift_args
91
92  printf_char: # deal with a %c:
93       sb $s1, 0($s6) # fill the buffer in with byte $s1,
94       sb $0, 1($s6) # and then a null.
95       move $a0, $s6 # and then do a print_str syscall
96       li $v0, 4 # on the buffer.
97       syscall
98       b printf_shift_args # branch to printf_shift_args
99
100 printf_perc: # deal with a %: # (this is redundant)
101       li $s5, ‘%’ # fill the buffer in with byte %,
102       sb $s5, 0($s6) # and then a null.
103       move $a0, $s6 # and then do a print_str syscall
104       li $v0, 4 # on the buffer.
105       syscall
106       b printf_loop # branch to printf_loop
107
108 printf_end: # restore the prior environment:
109       lw $ra, 32($sp) # release the stack frame.
110       lw $fp, 28($sp)
111       lw $s0, 24($sp)
112       lw $s1, 20($sp)
113       lw $s2, 16($sp)
114       lw $s3, 12($sp)
115       lw $s4, 8($sp)
116       lw $s5, 4($sp)
117       lw $s6, 0($sp)
118       addu $sp, $sp, -36 # return.
119       jr $ra
120
121 .data
122 printf_buf: .space 2
## end of printf.asm
This program is described in section 7.10.1.3.

This is a (somewhat) optimized version of a program which computes Fibonacci numbers. The optimization involves not building a stack frame unless absolutely necessary. I wouldn’t recommend that you make a habit of optimizing your code in this manner, but it can be a useful technique.

```assembly

## Daniel J. Ellard -- 02/27/94
## fib-o.asm-- A program to compute Fibonacci numbers.
## An optimized version of fib-t.asm.
## main--
## Registers used:
## $v0 - syscall parameter and return value.
## $a0 - syscall parameter-- the string to print.
.text
main:
subu $sp, $sp, 32 # Set up main’s stack frame:
sw $ra, 28($sp)
sw $fp, 24($sp)
addu $fp, $sp, 32

## Get n from the user, put into $a0.
li $v0, 5 # load syscall read_int into $v0.
syscall # make the syscall.
move $a0, $v0 # move the number read into $a0.
jal fib # call fib.

move $a0, $v0
li $v0, 1 # load syscall print_int into $v0.
syscall # make the syscall.

la $a0, newline
li $v0, 4
syscall # make the syscall.

li $v0, 10 # 10 is the exit syscall.
syscall # do the syscall.

## fib-- (hacked-up caller-save method)
## Registers used:
## $a0 - initially n.
## $t0 - parameter n.
## $t1 - fib (n - 1).
```
## FIB-O.ASM

```assembly
## define FIB-O.ASM:

# compute FIB (n - 2):
add $v0, $t1, $t2

## data for fib-o.asm:

.newline: .asciiz "\n"

## end of fib-o.asm
```
This program is outlined in section 7.11. The treesort algorithm is given in algorithm 7.4 (shown on page 245).

### treesort.asm

```mips
1 ## Daniel J. Ellard -- 03/05/94
2 ## tree-sort.asm -- some binary tree routines, in MIPS assembly.
3 ##
4 ## The tree nodes are 3-word structures. The first word is the
5 ## integer value of the node, and the second and third are the
6 ## left and right pointers.
7 ## &&& NOTE-- the functions in this file assume this
8 ## &&& representation!
9
10 ## main --
11 ## 1. Initialize the tree by creating a root node, using the
12 ## sentinel value as the value.
13 ## 2. Loop, reading numbers from the user. If the number is equal
14 ## to the sentinel value, break out of the loop; otherwise
15 ## insert the number into the tree (using tree_insert).
16 ## 3. Print out the contents of the tree (skipping the root node),
17 ## by calling tree_print on the left and right
18 ## children of the root node.
19 ##
20 ## Register usage:
21 ## $s0 - the root of the tree.
22 ## $s1 - each number read in from the user.
23 ## $s2 - the sentinel value (right now, this is 0).
24 .text
25 main:
26 li $s2, 0 # $s2 = the sentinel value.
27
28 ## Step 1: create the root node.
29 ## root = tree_node_create ($s2, 0, 0);
30 move $a0, $s2 # val = $s2
31 li $a1, 0 # left = NULL
32 li $a2, 0 # right = NULL
33 jal tree_node_create # call tree_node_create
34 move $s0, $v0 # and put the result into $s0.
35
36 ## Step 2: read numbers and add them to the tree, until
37 ## we see the sentinel value.
38 ## register $s1 holds the number read.
39 input_loop:
```
li     $v0, 5          # syscall 5 == read_int.
syscall
move  $s1, $v0        # $s1 = read_int
beq   $s1, $s2, end_input  # if we read the sentinel, break.

# tree_insert (number, root);
move  $a0, $s1        # number= $s1
move  $a1, $s0        # root = $s0
jal    tree_insert    # call tree_insert.
b     input_loop    # repeat input loop.
end_input:

## Step 3: print out the left and right subtrees.
lw     $a0, 4($s0)     # print the root’s left child.
jal    tree_print
lw     $a0, 8($s0)     # print the root’s right child.
jal    tree_print
b     exit             # exit.

## end of main.

## tree_node_create (val, left, right): make a new node with the given val and left and right descendants.
## Register usage:
## $s0 - val
## $s1 - left
## $s2 - right

subu   $sp, $sp, 32    # set up the stack frame:
sw     $ra, 28($sp)
sw     $fp, 24($sp)
sw     $s0, 20($sp)
sw     $s1, 16($sp)
sw     $s2, 12($sp)
sw     $s3, 8($sp)
addu   $fp, $sp, 32    # grab the parameters:
move  $s0, $a0        # $s0 = val
move  $s1, $a1        # $s1 = left
move  $s2, $a2        # $s2 = right
chapter 9. MIPS Assembly Code Examples

85    li   $a0, 12  # need 12 bytes for the new node.
86    li   $v0, 9   # sbrk is syscall 9.
87    syscall
88    move $s3, $v0
89
90    beqz $s3, out_of_memory  # are we out of memory?
91
92    sw   $s0, 0($s3)  # node->number = number
93    sw   $s1, 4($s3)  # node->left = left
94    sw   $s2, 8($s3)  # node->right = right
95
96    move $v0, $s3  # put return value into v0.
97    # release the stack frame:
98    lw    $ra, 28($sp)  # restore the Return Address.
99    lw    $fp, 24($sp)  # restore the Frame Pointer.
100   lw    $s0, 20($sp)  # restore $s0.
101   lw    $s1, 16($sp)  # restore $s1.
102   lw    $s2, 12($sp)  # restore $s2.
103   lw    $s3, 8($sp)  # restore $s3.
104   addu $sp, $sp, 32  # restore the Stack Pointer.
105   jr    $ra  # return.
106  ## end of tree_node_create.
107
108 ## tree_insert (val, root): make a new node with the given val.
109 ## Register usage:
110    ## $s0 - val
111    ## $s1 - root
112    ## $s2 - new_node
113    ## $s3 - root->val (root_val)
114    ## $s4 - scratch pointer (ptr).
115    tree_insert:
116        # set up the stack frame:
117    subu $sp, $sp, 32
118    sw    $ra, 28($sp)
119    sw    $fp, 24($sp)
120    sw    $s0, 20($sp)
121    sw    $s1, 16($sp)
122    sw    $s2, 12($sp)
123    sw    $s3, 8($sp)
124    sw    $s3, 4($sp)
125    addu $fp, $sp, 32
126    # grab the parameters:
127    move $s0, $a0  # $s0 = val
128    move $s1, $a1  # $s1 = root
9.9. TREESORT.ASM

130
131        # make a new node:
132        # new_node = tree_node_create (val, 0, 0);
133        move $a0, $s0 # val = $s0
134        li $a1, 0 # left = 0
135        li $a2, 0 # right = 0
136        jal tree_node_create # call tree_node_create
137        move $s2, $v0 # save the result.
138
139        ## search for the correct place to put the node.
140        ## analogous to the following C code:
141        ## for (;;) {
142        ##    root_val = root->val;
143        ##    if (val <= root_val) {
144        ##        ptr = root->left;
145        ##        if (ptr != NULL) {
146        ##            root = ptr;
147        ##            continue;
148        ##        }
149        ##        else {
150        ##            root->left = new_node;
151        ##            break;
152        ##        }
153        ##    }
154        ##    else {
155        ##        /* the right side is symmetric. */
156        ##    }
157        ##}
158
159        ## Commented with equivalent C code (you will lose many
160        ## style points if you ever write C like this...).
161
162        search_loop:
163        lw $s3, 0($s1) # root_val = root->val;
164        ble $s0, $s3, go_left # if (val <= s3) goto go_left;
165        b go_right # goto go_right;
166
167        go_left:
168        lw $s4, 4($s1) # ptr = root->left;
169        beqz $s4, add_left # if (ptr == 0) goto add_left;
170        move $s1, $s4 # root = ptr;
171        b search_loop # goto search_loop;
172
173        add_left:
174        sw $s2, 4($s1) # root->left = new_node;
175        b end_search_loop # goto end_search_loop;
CHAPTER 9. MIPS ASSEMBLY CODE EXAMPLES

175
176  go_right:
177    lw   $s4, 8($s1)  # ptr = root->right;
178    beqz $s4, add_right  # if (ptr == 0) goto add_right;
179    move $s1, $s4   # root = ptr;
180    b      search_loop   # goto search_loop;
181
182  add_right:
183    sw   $s2, 8($s1)  # root->right = new_node;
184    b      end_search_loop  # goto end_search_loop;
185
186  end_search_loop:
187
188  # release the stack frame:
189    lw   $ra, 28($sp)  # restore the Return Address.
190    lw   $fp, 24($sp)  # restore the Frame Pointer.
191    lw   $s0, 20($sp)  # restore $s0.
192    lw   $s1, 16($sp)  # restore $s1.
193    lw   $s2, 12($sp)  # restore $s2.
194    lw   $s3, 8($sp)   # restore $s3.
195    lw   $s4, 4($sp)   # restore $s4.
196    addu $sp, $sp, 32  # restore the Stack Pointer.
197    jr    $ra       # return.
198  ## end of node_create.
199
200  ## tree_walk (tree):
201  ## Do an inorder traversal of the tree, printing out each value.
202  ## Equivalent C code:
203  ## void tree_print (tree_t *tree)
204  ## {  
205  ##     if (tree != NULL) {
206  ##         tree_print (tree->left);
207  ##         printf ("%d\n", tree->val);
208  ##         tree_print (tree->right);
209  ##     }  
210  ## }
211  ## Register usage:
212  ## s0 - the tree.
213  tree_print:
214  # set up the stack frame:
215    subu $sp, $sp, 32
216    sw   $ra, 28($sp)
217    sw   $fp, 24($sp)
218    sw   $s0, 20($sp)
219    addu $fp, $sp, 32
move $s0, $a0  # grab the parameter:
beqz $s0, tree_print_end  # if tree == NULL, then return.
lw $a0, 4($s0)  # recurse left.
jal tree_print
lw $a0, 0($s0)  # print the value, and
li $v0, 1
syscall
la $a0, newline  # also print a newline.
li $v0, 4
syscall
lw $a0, 8($s0)  # recurse right.
jal tree_print
lw $ra, 28($sp)  # clean up and return:
lw $fp, 24($sp)  # restore the Frame Pointer.
lw $s0, 20($sp)  # restore $s0.
addu $sp, $sp, 32  # restore the Stack Pointer.
jr $ra  # return.
tree_print_end:  # end of tree_print.
out_of_memory:  # clean up and return:
la $a0, out_of_mem_msg
li $v0, 4
syscall
exit  # end of out_of_memory.
exit:  # end of program!
li $v0, 10  # 10 is the exit syscall.
syscall  # end of exit.
## Here's where the data for this program is stored:

```
.data
newline: .asciiz "\n"
out_of_mem_msg: .asciiz "Out of memory!\n"
```

## end of tree-sort.asm
Chapter 10

C for Programmers

by Bob Walton and Abolade Gbadegesin

This chapter is a quick tutorial in the C programming language for students who already know another language fairly well, such as Pascal, Fortran, or maybe even Scheme.

The student should acquire a book such as

Brian W. Kernighan and Dennis M. Ritchie,
*The C Programming Language*,

The student should not expect to learn C without writing some programs. We will give a few exercises, and there are more in the above text. It may also help the student to try things out using codecenter, which is a C interpreter.

Our method is to first present a C program, then explain the aspects of C one needs to know to understand the program, and lastly to restate these aspects in a more general fashion not tied to the particular program. We then present several refinements of the program, and another program, explaining further aspects of C as we go.

10.1 A Binary Search Program Prototype

First we will look at a C program that does something slightly non-trivial: it searches for an integer in a sorted list of integers, using the binary search algorithm, which is very high speed.

The essential idea of the binary search algorithm is to keep throwing away half of what is left of the list, making sure the number is not in the half we throw away. We start with the whole list.
We end with a single element part of the list, and then just check whether the number equals the single element.

Because the list is sorted, we can divide what is left of the list in the middle, and quickly find out which half we can throw away. If our number is less than the middle element, we throw the upper half away, and if it is greater than or equal to the middle, we throw the lower half away. We arrange things so the middle element itself is the least element of the upper half.

The first paragraph of this section defined the problem to be solved. This is called “specifying” the problem. The remaining paragraphs above gave an algorithm to solve the problem. This is called “preliminary design” of the program. Next we will give the design of the program interfaces, called “detail design”, then the code of the program itself, and lastly we will show how the program is tested and give test results.

### 10.1.1 Binary Search Detailed Design

The purpose of a detailed design is to decide on interfaces. We will write our detailed designs as code in which everything not relevant to determining interfaces has been replaced by English or mathematical statements in curly brackets { }.

The following is the detailed design of everything but the **binary_search** subroutine proper.

/* Prototype Binary Search Program

** File: bsearch1.c
** Author: course
** Version: 1
**
*/

/* This program determines whether a number is a member of
** a sorted list of numbers by using the binary search
** algorithm.
**
** This version of the program is a first rough prototype to
** see if basic ideas work out.
*/
#include <stdio.h> /* Printf. */

#define LISTSIZE 10

/* This list must be sorted in increasing order. */
int list[LISTSIZE] = { 350, 790, 1130, 1240, 1790,
                      2350, 2680, 3240, 3790, 4350
                    };

/* Function to return the index (0, 1, ...) of the element in the list containing the number. Return -1 if the number is not in the list. The number of elements in the list, or length, must be given. */
int binary_search (int number, int list[], int length);

int main ()
{
    { Determine if 3240 is in the list and print answer. }
    { Determine if 3241 is in the list and print answer. }

    return 0;
}

Since the detailed design is written in the C language, except for the English in curly brackets, this is a good place to start explaining C. Starting from the top of the design, the relevant facts about C are:

1. All characters between the strings “/ *” and “* /” are ignored: this is the way to put comments into C source code.

2. Lines beginning with the character “#” are directives to the C preprocessor.

3. #include <stdio.h> directs the preprocessor to read the contents of the file stdio.h into the source at the line where the directive appears. This file contains declarations for functions such as printf used by this program. The name stdio abbreviates “standard input-output package”.

4. #define directs the preprocessor to record a macro definition. The syntax is:
#define <macro-name> <macro-definition>

All instances of identifier <macro-name> in the source code are then replaced with <macro-definition>.

5. Identifiers are used to name macros, variables, functions, types, etc. Identifiers may contain letters or digits, beginning with a letter. The underscore character “_” is treated as a letter.

6. Some identifiers are keywords, and cannot name variables, etc. Some of the keywords are names for builtin number types: e.g. int for integers. Others name statements, like if and return.

7. Semicolons “;” are used to end declarations and statements.

8. The form for a simple variable declaration is:

   <type-name> <variable-name>;

E.g.:

   int number;
   int length;

9. The form for a complex variable declaration is:

   <type-name> <declaration-expression>;

E.g.:

   int list[10];

Here the <declaration-expression> is, almost, an expression using the variable being declared. Since list is a vector of ints, its typical use is an expression such as list[i] which evaluates to an int. The <declaration-expression> is just this expression with i replaced by the size of the vector, i.e. 10. Note that LISTSIZE is a macro replaced by 10.

10. A variable can be initialized by adding an initialization expression to the variable’s declaration:

    <type-name> <declaration-expression> = <initialization-expression>;
10.1. A BINARY SEARCH PROGRAM PROTOTYPE

E.g.:

```
int x = 10;
```

11. The `<initialization-expression>` for a vector can be a curly bracketed sequence of element initial values, as in

```
int list[LISTSIZE] = { 350, 790, 1130, 1240, 1790, 2350, 2680, 3240, 3790, 4350 };
```

12. Subscripting is zero-based: the first element of `list` is `list[0]`.

13. Before a function can be used, a declaration must appear for it of the form:

```
<type-name> <declaration-expression>;
```

The `<declaration-expression>` has the syntax of a call on the function with the arguments replaced by their declarations, which are essentially variable declarations. The `<type-name>` specifies the returned type. E.g.:

```
int binary_search (int number, int list[], int length);
```

14. If the size of the first dimension of an array is not known, it may be omitted in array variable declaration. E.g. `list[]` above. This typically happens in function argument declarations. The missing size must be passed as a separate argument to the function: e.g. `int length`.

15. When a C program executes, the operating system loads the program and calls the function named `main`. To end the program, that function returns an integer which is zero if there is no error, and an error code otherwise.

16. A sequence of program statements surrounded by curly brackets `{ }` is called a program block. A program block can be used anywhere a statement can be used.

17. A function definition (e.g. that of `main` above) is just like a function declaration with the semi-colon replaced by a program block containing the statements executed by the function.

18. The `return` statement may be used in a function to quit the function and designate the value returned from the function. It has the form:

```
return <returned-value-expression>;
```
To determine global interfaces, we do not have to do any detailed design of the `binary_search` function internals. But it would be nice to decide how we are going to represent the “remaining list” at each stage of the computation, so we give the following:

```c
int binary_search (int number, int list[], int length)
{
    int low = 0;
    int high = length;
    int middle; /* = (low + high) / 2 */

    { Remaining list is all list[j] for low <= j < high. }

    { Execute binary search algorithm and return answer. }
}
```

To understand this we need a bit more information about C:

19. Variables are either global (outside any function) or local (inside a `{ }` block). Local variables are declared after the beginning bracket, `{`, of a block, and can be used only inside that block.

20. Global variables are declared anywhere outside a function, and can be accessed by any function that is defined after the declaration of the variable.

21. The initialization expression of a local variable can be anything computable at the spot the expression appears.

22. The initialization expression of a global variable can only contain constants and operators; and not variables or function calls.

23. Function arguments are like local variables declared at the very beginning of a function `{ }` block, and are given the argument values passed by the call to the function as initial values.

### 10.1.2 Binary Search Code

To make the code from the detailed design, we replace the `main` and `binary_search` subroutines. First `main`:
int main ()
{
    /* Determine if 3240 is in the list: */
    printf ( "binary_search (3240, list, LISTSIZE) = %d\n",
             binary_search (3240, list, LISTSIZE));

    /* Determine if 3241 is in the list: */
    printf ( "binary_search (3241, list, LISTSIZE) = %d\n",
             binary_search (3241, list, LISTSIZE));

    return 0;
}

The additional facts about C relevant to understanding the main code are:

24. The formatted print function printf, declared in stdio.h, is the standard way to produce output. Its first argument is a string, called the format string. The rest of the arguments are printed as directed by specifications in the format string that begin with %, such as %d.

25. A string constant is a sequence of character representatives between double quotes. Most characters are their own representatives.

26. The linefeed character is represented by \n in a character string.

27. printf prints the characters of its format string until it comes to a % specification. It then prints its next argument according to that specification. printf continues processing format characters and specifications till it gets to the end of the format string.

28. The format specification %d prints an int argument as a decimal integer.

Next binary_search:
int binary_search (int number, int list[], int length)
{
    int low = 0;
    int high = length;
    int middle; /* = (low + high) / 2 */

    /* The remaining list in the binary search algorithm is */
    /* the set of all list[j] such that low <= j < high. */
    /* Then it is true throughout that: */
    /* if number == list[j] then low <= j < high */

    while (high >= low + 2)
    {
        /* More than one element remaining */

        middle = (low + high) / 2;
        if (number < list[middle]) high = middle;
        else low = middle;
    }

    /* At most one element remaining */

    if (low < high && list[low] == number) return low;
    else return -1;
}

The additional facts about C relevant to understanding the binary_search function code are:

29. Expressions are built by combining constants, variables, and operators. Expressions have values, which are often numbers.

30. The standard operators +, -, *, and / are provided.

31. The relational operators == (equal), != (not equal), >, >=, <, and <= are provided.
10.1. A BINARY SEARCH PROGRAM PROTOTYPE

32. The boolean value “true” is represented by any non-zero value. The boolean value “false” is represented by any zero value. The relational operators evaluate to 0 (false) or 1 (true).

33. The equality operator == is sometimes mistyped as the assignment operator =. The assignment operator is also a binary operator, and such a typo may be legal C, and cause a nasty bug.

34. C has a while-loop statement of the form

   ```c
   while ( <condition-expression> ) <statement>
   ```

   where <statement> is often a { } block. This (1) evaluates <condition-expression> and terminates the while-loop if false (zero); otherwise it (2) evaluates the <statement> and loops back to (1).

35. C has an “if” statement of the form

   ```c
   if ( <condition-expression> ) <statement1>
   else <statement2>
   ```

   where the else part can be omitted.

10.1.2.1 Binary Search Test

The program we have written, in the form of the file bsearch1.c, can be compiled and tested by the following Makefile commands:

```makefile
bsearch1:  
bsearch1    
gcc -g -ansi -pedantic -o bsearch1 bsearch1.c
```

```makefile
bsearch1.out:  
bsearch1
bsearch1 >bsearch1.out 2>&1
```

The C compiler we use is gcc. This particular compiler accepts many C dialects, so the options -ansi -pedantic are used to restrict the language it accepts to correct ANSI C.

The -g option directs the compiler to produce extra information in the binary output file to permit the program to be debugged with a typical symbolic debugger, such as gdb.

The -o bsearch1 option names the file which will contain the compiled program.

The bsearch1.c filename is the name of the source code file.

The command that makes bsearch1.out merely runs the program and puts the output in the bsearch1.out file. The expression 2>&1 at the end is Bourne shell (sh) notation meaning
that the error output (denoted by the file descriptor 2) is to be merged with the standard output (denoted by the file descriptor 1).

The contents of bsearch1.out, once it has been made, are:

binary_search (3240, list, LISTSIZE) = 7
binary_search (3241, list, LISTSIZE) = -1

10.2 The Basics of C

We have described enough C to understand the bsearch1 prototype binary search program. We are now going to repeat that description of C, but in a more abstract and general way, more like a language reference manual. We will deal in order with syntax, data, expressions, functions, statements, and preprocessor commands (#define and #include). We are only describing the basics of the C language here; other features will follow later when we examine other programs.

10.2.1 Syntax

R–1. Spaces, tabs, and line-ends all are equivalent in C if they are not inside quoted character string constants or character constants.

R–2. Comments begin with the string “/*” and end with the string “*/”.

R–3. A comment may include ends of lines, and some pretty horrific bugs have involved forgetting the “*/” at the end of a line and accidentally commenting out the subsequent line.

R–4. Comments cannot be nested.

R–5. Identifiers, which are used to name variables, constants, types, and functions, consist of letters and digits, and must begin with a letter.


R–7. The underscore “_” is treated as a letter, and may be used in and begin an identifier.

R–8. Some implementations use only the first 31 characters of an identifier to determine identifier uniqueness.

R–9. Certain C identifiers are reserved for use as keywords:
These identifiers cannot be used as names of variables, constants, types, or functions.

R–10. Integers and floating point numbers may be written as in other languages as strings of digits, a possible decimal point, and a possible exponent part; e.g. \(1.0\times10^{-9}\).

R–11. Numbers are unsigned: to get a signed number you must use an arithmetic expression involving the sign operator.

R–12. Any place C expects a number constant, you can also use an arithmetic expression that contains no variables or function calls. E.g. you can say an array length is \(20 - 5\) instead of saying it is 15.

R–13. A non-zero integer beginning with the digit 0 is octal. I.e. the integers 070 and 56 are the same.

R–14. A character is an integer for purposes of computation in arithmetic expressions. The correspondence between characters and integers, for all common American computers today, is defined by the American Standard Code for Information Interchange, or ASCII. In a UNIX system the file /usr/pub/ascii contains this correspondence. Here is part of that file:
R–15. The number value of a character \( x \) can be denoted in C by enclosing the character in single quotes, as in ‘\( x \)’, which is the same as the integer 0170 (octal) (see table above). Such single quoted characters are called character constants.

R–16. Strings of characters are represented by enclosing them in double quotes; e.g. "Hello there!". These are called string constants.

R–17. If a string constant is too long to fit on one line, it may be broken into two or more consecutive string constants that will be joined together by the C compiler. E.g.

"Hello There!\n"
"I’m Bob.\n"

is equivalent to "Hello There!\nI’m Bob.\n."

R–18. Not all characters can be denoted by themselves in a character or string constant. Some are denoted by a backslash (\) followed by a letter or the character being denoted:

\n denotes a line feed (end of line)
\t denotes a tab
\f denotes a form feed
\ \ denotes a backslash
\' denotes a single quote
\" denotes a double quote

E.g. ‘\’’ is the character constant for ’ and ”Hi!\n” is a string constant containing a line feed.

R–19. Any character can be represented by a backslash followed by its integer code in octal. \0 is commonly used to represent the NUL ASCII character, which is used to end character strings in memory.

R–20. Statements are ended by semicolons (;). (Unlike Pascal, semicolons end statements in C, instead of separating them.)

R–21. The curly brackets { } are used to surround statement sequences in order to make “blocks” (blocks are also called compound statements). A block can be used anywhere a statement can; e.g.:

    if (x == 10) x = 88;
    if (x == 9) { x = 55; y = 77; }

Note there is no semicolon after the }.

R–22. Blocks do not have values and cannot be used where expressions are expected.

R–23. Expressions are built from operators such as + and * in the normal way.

R–24. An expression followed by a semicolon is one form of statement. Any value of the expression is discarded.

R–25. The assignment operator = is a binary operator; e.g. x = 9 is an expression and not just a statement.

10.2.2 Data

R–26. A simple variable declaration can have the form:

    <type-name> <variable-name>;

The `<type-name>` can be `int`, `float`, `char`, etc.

R–27. There are several sizes of integers and floating point numbers:

<table>
<thead>
<tr>
<th>Type</th>
<th>DEC MIPS</th>
<th>IBM PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>long</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

R–28. Integer types are `char`, `short`, `int`, and `long`. The `int` type is the “natural” size of integers on given hardware. You may also write `long int` and `short int` which mean the same as `long` and `short`, respectively.

R–29. Integer types are signed by default, except for `char`. To make them unsigned put `unsigned` in front of them: e.g. `unsigned int`. The type `unsigned` by itself means `unsigned int`.

R–30. To make `char` signed put `signed` in front of it. The `char` type may be either signed or unsigned depending on the hardware.

R–31. The `char` type is suitable for holding characters and booleans (0 or 1).

R–32. You can give a variable an initial value when you declare it by writing “= `<initial-value-expression>`” at the end of the variable’s declaration:

```
<type-name> <variable-name> = <initial-value-expression>;
```

Any expression that can be computed at the spot where the variable is declared may be used for local variable (see below), but only constant expressions may be used for global variables (see below).

R–33. A local variable is one declared just after the `{` in a block; e.g.:

```
{    int x = 0;
    int y = x + 1;
    . . . . .
}
```

A local variable exists from the point it is declared until the end of its block.
10.2. **THE BASICS OF C**

R–34. Local variables without initialization expressions are initialized to random values. This can cause bugs.

R–35. A global variable is one declared outside all blocks; e.g.:

```c
int x;
int y = 0;
int z = 8 + 9;
```

R–36. For global variables, initialization expressions can contain only constants and operators: no variables or function calls are allowed. Global variables without initialization expressions are initialized to zero.

R–37. Variables with more complex types have declarations of the form:

```c
<type-name> <declaration-expression>;
```

Here the `<declaration-expression>` is, almost, an expression that uses the variable and produces a value of type `<type-name>`. Examples:

```c
int v[10];
float m[10][30];
int (*(f[10]))(int j);
```

Here `v` is a vector of 10 integers which would be used in expressions such as `v[i]`, `m` is a `10 \times 30` matrix of integers which would be used in expressions such as `m[i][j]`, and `f` is a vector of pointers to functions that are called with an integer argument and return an integer value, and which would be used in expressions such as `(*(f[i]))(j)`.

R–38. Subscripting is zero-based: the first element of `v` is `v[0]`.

R–39. For declaration expressions of arrays, subscripts are replaced by dimension sizes.

R–40. The first dimension size in an array declaration may be omitted, if it is not known, as in:

```c
char line[];
int m[][10];
```

Dimension sizes are most often unknown for function arguments.

R–41. Dimension sizes (other than a missing first dimension size) must be integer constants, and cannot contain variables, even for arguments of functions, unlike Fortran.
R–42. For declaration expressions of functions, arguments are replaced by declarations of the arguments. Argument names may be included or omitted.

R–43. Several variables or functions may be declared in one declaration statement by using the form:

\[
<\text{type-name}> <\text{declaration-expression}_1>, <\text{declaration-expression}_2>, \ldots;
\]

or

\[
<\text{type-name}> <\text{declaration-expression}_1> = <\text{init-expression}_1>, <\text{declaration-expression}_2> = <\text{init-expression}_2>, \ldots;
\]

For example,

\[
\text{int } x = 0, y = 7, z, (*f[10]))(\text{int});
\]

R–44. Initialization expressions for arrays may have element values in curly brackets:

\[
\text{int } v[5] = \{ 1, 3, 5, 7, 9 \};
\]
\[
\text{int } m[2][3] = \{ \{ 11, 12, 13 \}, \{ 21, 22, 23 \} \};
\]

R–45. Character strings are represented by vectors of \text{char}s. E.g., a line of text could be stored in:

\[
\text{char } line[100];
\]

R–46. The last used character of a string is immediately followed by the ASCII NUL character, numeric code 0, which indicates the end of the string. Either 0 or \text{	extquoteleft\textbackslash 0} may be used to denote the NUL character. The above \text{line} vector variable will actually hold only 99 characters plus the ending NUL.

R–47. Initialization expressions for arrays of characters may use string constants as equivalents to bracketed lists of character constants. E.g. \text{"xy"} is equivalent to \{ \text{\textquoteleft}x\text{\textquoteleft}, \text{\textquoteleft}y\text{\textquoteleft}, \text{\textquoteleft}\textbackslash 0\text{\textquoteleft}} \} when used as an initializer for a vector of \text{char}s.

R–48. Note that double quoted string constants always have an implicit \text{	extquoteleft\textbackslash 0\textquoteleft} at their end.

R–49. If an array initialization expression ends early, the rest of the elements are initialized to zero.
10.2. THE BASICS OF C

R–50. If an array variable is declared with an initial value, the first dimension of the declaration may be omitted and is determined from the initial value. E.g.:

char message[] = "This is an IMPORTANT MESSAGE!";

10.2.3 Expressions

R–51. Expressions are built by combining constants, variables, and operators.

R–52. Expressions have values; statements do not.

R–53. The standard arithmetic operators, +, −, *, and / are supported. There is no exponentiation operator. There is an operator % that computes remainders.

R–54. The standard arithmetic operators obey the usual precedence rules.

R–55. The assignment operator = is a binary operator. The value of $x = y$ is the value of $x$ after the assignment.

R–56. Any expression can be used in an if statement as a conditional expression. Any non-zero value means “true”; any zero value means “false”.

R–57. The comparison operators are ==, != (not equal), <=, <, >=, >.

R–58. A common bug is to type = instead of ==. Sometimes the result is legal C and causes great problems in debugging the program. E.g. $x = y$ is likely to be always true and set $x$ equal to $y$ to boot.

R–59. The boolean operators, that operate on truth values, are && (and), || (or), and ! (not). These return 0 (false) or 1 (true).

(There are a rather different set of bitwise integer operators &, |, and ~ that should not be confused with the boolean operators).

R–60. In $x$ && $y$, if $x$ is false, $y$ is not evaluated. In $x$ || $y$, if $x$ is true, $y$ is not evaluated. This is called “short circuiting”.

R–61. There is an operator += such that $x += y$ is shorthand for $x = x + y$. Similarly there are operators -=, *=, /=, and %=.

R–62. The if statement cannot be used as an expression. Instead there are operators ? and : that are used together in expressions such as $x$ ? $y$ : $z$ that mean “if $x$ then the value of $y$ else the value of $z$".
R–63. The arithmetic operators have lower precedence than the comparison operators, which have lower precedence than the boolean operators, which have lower precedence than the ? : operators. The arithmetic operators have lower precedence than the assignment operators =, +=, -=, etc. It is best to ignore other C operator precedences and use parentheses to indicate order of evaluation instead.

### 10.2.4 Functions

R–64. A function must be declared before it is used. See the section on *Data* for details of function declarations.

R–65. A function definition is just like the declaration but with the semi-colon replaced by a curly bracketed ({} ) block.

R–66. Function arguments are like local variables declared at the very beginning of a function block, and are given the argument values passed by the call to the function as initial values.

R–67. Resetting function arguments in a function has no effect on the caller of the function, as the arguments are variables local to the called function. This is referred to as “call by value”.

R–68. A function that does not return a value should be declared as if it had a value of the `void` type; e.g.

```c
void sort (int list[], int length);
```

R–69. Functions that take no arguments must be declared as if they had a single `void` argument; e.g.:

```c
long time_of_day (void);
```

This is for historical reasons; if the `void` were omitted the declaration is interpreted as giving no information about the number and types of the arguments.

R–70. The `return` statement terminates a function and specifies the value to be returned by the function. This statement has the form:

```c
return <returned-value-expression>;
```

R–71. The program itself is defined as a function called `main`. When the program runs, this function is called.
R–72. The main function returns an integer program status code, that is non-zero if the program detected an error, and 0 if there was no error.

## 10.2.5 Statements

R–73. The if statement has the forms:

\[ \text{if ( <condition-expression> ) <statement>} \]

and

\[ \text{if ( <condition-expression> ) <statement1> else <statement2>} \]

R–74. A while-loop has the form:

\[ \text{while ( <condition-expression> ) <statement>} \]

This (1) evaluates <condition-expression> (2) terminates if that is false, and otherwise (3) evaluates the <statement> and loops to (1).

R–75. A curly bracketed block may be used anywhere a statement may be:

\[
\text{if (x == 0)}
\{ \\
\text{\hspace{1cm} x = 10;} \\
\text{\hspace{1cm} y = y + 1;} \\
\}
\]

\[
\text{while (i != 0) sum += v[i = i - 1];}
\]

\[
\text{while (x != 0)}
\{ \\
\text{\hspace{1cm} y = y * 9;} \\
\text{\hspace{1cm} x = x - 1;} \\
\}
\]

R–76. A do-while-loop has the form:
do <statement> while ( <condition-expression> )

This (1) evaluates <statement>, (2) evaluates <condition-expression> (3) terminates if that is false, and otherwise loops to (1).

### 10.2.6 Preprocessing

R–77. Lines beginning with the character “#” are C preprocessor directives.

R–78. Preprocessor directives do not end with a semi-colon.

R–79. Directives such as:

```c
#include <stdio.h>
#include "local.h"
```

direct the preprocessor to read the contents of the files `stdio.h` and `local.h` into the source at the line where the directive appears.

R–80. If an `#include` filename is surrounded by angle brackets `< >`, the filename is relative to some public directory.

On UNIX systems this directory is `/usr/include`.

R–81. If an `#include` filename is surrounded by double quotes " ", the filename is relative to the directory containing the file that contains the `#include` directive.

R–82. Directives of the form:

```c
#define <macro-name> <macro-definition>
```

cause all occurrences of the identifier `<macro-name>` to be replaced by `<macro-definition>`. The latter may be any sequence of identifiers, operators, parentheses and brackets, constants, etc. For example:

```c
#define LINESIZE 100
```

R–83. It is a C convention to define compile time constants by macros using identifiers in which all letters are capitalized.
10.3 Adding Input to the Binary Search Prototype

Next we will add to our prototype binary search program the ability to input the number to be searched for. This involves changing just the main function. The goal is to permit the user to type numbers at the program, one per line, and search for each number, printing the results.

This is a case where having separate preliminary and detailed designs makes little sense. We will go straight to what we shall call the detailed design of the main function, which will emphasize identifying the standard C subroutine library functions to be called.

The detailed design for our new main is:

```c
int main ()
{
    /* Buffer to read input line into. Leaves
     ** plenty of room for the usual 80 column line.
     */
    char line [100];
    int number;

    { Print:
        Type in numbers to be searched for, one per line.
        End with control-D.
    }

    while (not end of input))
    {
        { Read a line from the input and read number from
          line. If line does not contain a number, print
          an error message including the undecipherable line,
          and continue with the next iteration of the while
          loop.
        }

        { Call binary_search with number and list. }

        { Print result of call. }
    }

    return 0;
}
```
There is little new about C here, so we will wait until after we see the code to present any more facts about C. The code for main is:

```c
int main ()
{
    /* Buffer to read input line into. Leaves 
    ** plenty of room for the usual 80 column line. 
    */
    char line [100];
    int number;

    printf ( "Type in numbers to be searched for, "
            "one per line.\n"
            "End with control-D.\n"
            "\n" );

    while (gets (line))
    {
        if ( 1 != sscanf (line, "%d", &number))
        {
            printf ("CANNOT UNDERSTAND %s\n", line);
            continue;
        }

        printf
        ( "binary_search (%d, list, LISTSIZE) = %d\n",
            number,
            binary_search (number, list, LISTSIZE) );
    }

    return 0;
}
```

The new facts about C that you need to understand this code are:

36. Character strings are stored in vectors of `char`s. E.g. the `line` variable above.

37. The function `gets` reads a line from the standard input (usually the console) and writes it into its first argument, which is string (vector of `char`s).

38. An input line is a sequence of characters terminated by a newline character: `\n`. 
39. When `gets` reads an input line, it replaces the newline `\n` with a NUL character `\0`. The NUL character has ASCII value 0.

40. The `gets` function returns true if it reads a line, and false if it does not. Assuming the input is coming from a successfully opened input file, the only normal reason (barring hardware error) for `gets` to return false is that the input has reached the end of the input file. So users of `gets` often assume that the function returns false only at the end of the input file.

41. The `gets` function is about the only platform-independent input function. All others have platform-dependent treatments of end of file or end of line. Thus for machine-independent code one is forced to read lines first, and then dissect them.

42. The `scanf` function is a sort of inverse `printf` which reads data (e.g. number) from some input. The `sscanf` function is a variant of `scanf` that reads characters from a string (i.e., vector of `char`s).

43. A call to `sscanf` has the form:

```
sscanf (<input-string>, <format-string>, &<variable-name>, ...)
```

The `sscanf` function can read what `printf` can write, more or less.

44. The operator `&`, followed by a variable or function name, evaluates to the address of that variable or function. The address of a variable is the location in memory where the variable’s value is stored.

45. The address of the variables must be supplied to `sscanf`, instead of the variables themselves, because in C arguments to functions are passed by value, not by reference. This means that to change the variable value, `sscanf` must know where it is stored instead of just its value, which is irrelevant anyway, since it is about to be changed.

46. In C, vector variables, such as `line`, in fact always represent the address of where the vector is stored, and never the actual value of the variable, which is why we have been able to avoid writing `&list` and can avoid writing `&line`.

47. When `sscanf` sees a `%d` in its format string, it reads the next characters in its input string expecting to see a decimal integer, and if it finds one, copies it into the location addressed by its next argument.

48. The `sscanf` function returns the number of values it successfully read. It stops, for example, if it cannot find a digit or sign when asked to read an integer.
49. When the `printf` function encounters a `%s`, it expects its next argument to be a character string, which it prints.

50. If the `continue` statement is executed anywhere in a loop, control passes (as for a `goto`) to the next iteration of the loop.

Now that we have the code, we can test it. We replace the `main` function in `bsearch1.c` to make `bsearch2.c`. Compiling `bsearch2.c` is just like compiling `bsearch1.c`. But testing is a bit different, because now we have input.

We could test by just executing the `bsearch2` program and typing things at it. But if we want to show our work in a repeatable way, it pays to make up an input file and a `Makefile` entry.

The `Makefile` command is:

```
make: bsearch2 bsearch2.in
    bsearch2 <bsearch2.in >bsearch2.out 2>&1
```

The input file, `bsearch2.in`, is (since comments are not allowed, it has no header):

```
-28
x
350
790
1132
1240
1790
2353
2680
3240
3799
4350
10000
```

And the output file, `bsearch2.out`, is:
Type in numbers to be searched for, one per line.
End with control-D.

binary_search (-28, list, LISTSIZE) = -1
CANNOT UNDERSTAND x
binary_search (350, list, LISTSIZE) = 0
binary_search (790, list, LISTSIZE) = 1
binary_search (1132, list, LISTSIZE) = -1
binary_search (1240, list, LISTSIZE) = 3
binary_search (1790, list, LISTSIZE) = 4
binary_search (2353, list, LISTSIZE) = -1
binary_search (2680, list, LISTSIZE) = 6
binary_search (3240, list, LISTSIZE) = 7
binary_search (3799, list, LISTSIZE) = -1
binary_search (4350, list, LISTSIZE) = 9
binary_search (10000, list, LISTSIZE) = -1

10.4 Using Pointers in the Binary Search Prototype

Now we will change the binary_search subroutine to use pointers to elements of the list, rather than indices of these elements.

First, we must explain what a pointer is. Conceptually, a list is stored in memory locations that are consecutive, that is, next to each other. Each location has an “address”, rather like each house on a block has an address, and the addresses of the elements of a list are like integers that increase in a regular way as we go through the list. In fact, to go from one element to the next, one simply adds the size of the element to the previous element’s address in order to get the next element’s address. All the elements of a list have the same size.

The size of an element may be measured in different units. The most common units are bits, bytes, and elements. The size of an element in elements is, of course, 1 element, and you may wonder why we should think of this at all, but in fact that is what the C language uses most of the time, as we shall see.

Let us assume that the list variable in our prototype program is stored in computer memory beginning at byte address 1000 (we will say what this means below), and look at the addresses at which the list elements are stored:
Consider the case where we are on an IBM PC. All the bits of memory may be numbered, 0, 1, 2, ...; The 8000'th bit is the first bit of the list element whose value is 350. This value consumes 16 bits, on an IBM PC. The C expression for the bit address 8000 is \texttt{list+0} (or just plain \texttt{list}). When we add just 1 to the C expression, to get the C expression \texttt{list+1}, we get a new bit address, 8016, that is 16 bits greater than the bit address of \texttt{list+0}. It is 16 bits greater because on an IBM PC an \texttt{int} value is 16 bits long.

We could play this game with byte addresses too. The C expression for the byte address 1000 on the IBM PC is \texttt{list+0}. The C expression \texttt{list+1} evaluates to the byte address 1002, on the IBM PC; because an \texttt{int} is 2 bytes long. However, on the DEC MIPS computer, \texttt{list+1} evaluates to the byte address 1004, because an \texttt{int} is 4 bytes long on this computer.

Now that we know what C expressions like \texttt{list+0} and \texttt{list+1} represent, consider what we can do with them. The unary \texttt{*} operator, when applied to a C pointer (i.e. a C expression representing an address), returns the value pointed at (i.e. the value addressed). Thus in the above example \texttt{* (list+0)} returns 350 and \texttt{* (list+1)} returns 790.

We can also use the \texttt{*} operator on the lefthand side of an assignment to change the value addressed. Thus \texttt{* (list+1) = 750} would change the second element from 790 to 750.

Lastly, the notation \texttt{list[n]} is completely equivalent in C to \texttt{* (list+n)}. Thus we have been using pointers all along and not known it.

Now lets go back to our programming. Just as before, we will do a brief detailed design of our \texttt{binary_search} subroutine in order to establish how we are going to represent the part of the list we have retained:
10.4. USING POINTERS IN THE BINARY SEARCH PROTOTYPE

```c
int binary_search (int number, int list[], int length)
{
    int * low = list;
    int * high = list + length;
    int * middle; /* = low + (high - low) / 2 */

    { Remaining list is all *p for low <= p < high. }

    { Execute binary search algorithm and return answer. }

}
```

The new facts about C that you need to understand this detailed design are:

51. A pointer is a value that is the address of a place to store another value in memory. A pointer is said to point at this other value. A pointer to a variable is the address of where the variable stores its value in memory. A pointer to a list element is the address of where the element value is stored in memory.

52. A pointer to a variable or list element allows one to change the variable or list element value, because you know what memory location to alter.

53. A pointer usually points at some value of a type known at compile time. Thus one has “pointer to int”, “pointer to float”, etc. We refer to the type of value pointed at as the target type of the pointer.

54. If a pointer \( p \) points at an element of a vector, the pointer \( p+1 \) points at the next element of the vector, and the pointer \( p-1 \) points at the previous element of the vector.

55. If a pointer \( p \) points at an element of a vector, and \( n \) is a non-negative integer, then the pointer \( q=p+n \) points at the \( n \)'th element of the vector after the element pointed at by \( p \). Also, the pointer \( r=p-n \) points at the \( n \)'th element of the vector before the element pointed at by \( p \).

56. Continuing the last example, the expression \( q-p \) is equal to \( n \), and \( r-p \) is equal to \( -n \). If you subtract two pointers that point at elements in the same array, the difference is the integer you would add to the second pointer to get the first pointer.

57. The names of vectors and arrays denote pointers. These names are constant expressions; they cannot be assigned values.
The code for binary_search with pointers is:

```c
int binary_search (int number, int list[], int length)
{
    int * low = list;
    int * high = list + length;
    int * middle; /* = low + (high - low) / 2   */

    /* The remaining list in the binary search algorithm is
      ** the set of all *p such that low <= p < high.
      **
      ** Then it is true throughout that if p is a pointer at
      ** an element of list:
      **
      **     if number == *p then low <= p < high
      */

    while (high >= low + 2)
    {
        /* More than one element remaining */
        middle = low + (high - low) / 2;
        if (number < *middle) high = middle;
        else low = middle;
    }

    /* At most one element remaining */

    if (low < high && *low == number) return low - list;
    else return -1;
}
```

The new facts about C that you need to understand this code are:

58. Two pointers at elements in the same list may be compared. $p < q$ if and only if $q$ points at an element in the list that is after, in memory, the element pointed at by $p$. This is the same as saying that $0 < q-p$.

59. The unary $*$ operator may be applied to a pointer to produce the value the pointer points at.
E.g., \(*p == 5\) tests whether the value pointed at by \(p\) equals 5.

60. The type of \(*p\) is the target type of \(p\). Thus in the expression \(*p == 5\), the target type of \(p\) must be some number type, such as \(\text{int}\).

61. The expression \(*p\) can also be used on the left side of the assignment operator \(=\), as in \(*p = 5\). This copies the value 5 into the memory pointed at by \(p\), so that thereafter \(*p\) will equal 5.

62. The expression \(p[n]\) is shorthand for \(* (p+n)\). That is, these two ways of referring to the \(n\)'th element after the element pointed at by \(p\) are completely equivalent.

If we replace the subroutine \(\text{binary\_search}\) in \(\text{bsearch2.c}\) we get a program \(\text{bsearch3.c}\) which compiles and tests identically to \(\text{bsearch2.c}\).

10.5 The Final Binary Search Program

We will now present the final \(\text{bsearch}\) program. This program adds to our last prototype program the ability to read in and sort the list of numbers.

Divide-and-conquer is one of the primary ways to organize a program: the program is modularized into subroutines. A main reason for doing this arises from the fact that programs are inherently untestable: you cannot make a program correct just by testing it against enough possible inputs, though testing is required to catch some kinds of bugs. Instead, you must also “prove” that the program is correct by reading it, a little like a mathematician proves a theorem. Its very hard to do this for big pieces of code, so instead one divides a large program into small pieces of code with well-defined interfaces that can be checked independently of one another. There are other benefits from such modularization; for example, the ease of which part of one program can be reused in other programs is increased.

Reading the code to find bugs is called “desk checking”. Many programmers find that using a debugger to single-step through a subroutine with typical data is a useful way of organizing desk checking.

The three subroutines in our final version of the \(\text{bsearch}\) program are the binary search subroutine, a subroutine to read in the list of numbers to be searched, and the main program. We also use a standard C library function, \(\text{qsort}\), to sort the list.

It is common when building such a multi-subroutine program to place each subroutine in a file of its own, and connect them together by a \(\text{header file}\), or .h file. The \(\text{bsearch.h}\) file for our program is:
/ * Binary Search Program Header File
**
** File: bsearch.h
** Author: course
** Version: 1
*/

/* This program determines whether a number is a member of a
** sorted list of numbers by using the binary search algorithm.
*/

/* Maximum number of elements in list to be searched.
*/
#define MAXIMUM_LIST_SIZE 1000

/* Data type of list to be searched. This is a vector of ints.
*/
typedef int * list_of_ints;

/* Function to read in a list_of_ints. The list is read, one
** integer per line, from the standard input. An empty line or
** an end of file (control-D for a terminal) terminates the list.
**
** This function allocates the list and returns it and its length.
** It also returns an error count equal to the number of lines
** that contained something other than a number at their beginning.
** These lines also printed in error messages by this function.
*/
void read_list
( list_of_ints * list,
  int * length,
  int * error_count );
/* Function to return the index (0, 1, ...) of the element in
** the list containing the number. Return -1 if the number is
** not in the list. The number of elements in the list,
** or length, must be given.
*/
int binary_search
( int number,
  list_of_ints list,
  int length );

This file introduces a data type, list_of_ints, that denotes a list of integers (below we will
describe typedef). A new subroutine, read_list, is introduced to read a list_of_ints. It
is convenient to limit the maximum number of integers read, though a more complex program could
handle lists of unlimited length. Because several subroutines might use this limit, and because a
user of the program might want to know about it and possibly increase it, the limit is denoted by
the macro MAXIMUM_LIST_SIZE placed at the very beginning of the header file.

There are a few new C constructs used in this header file:

63. Declarations prefixed by typedef declare new type names. The declaration is written as if
a variable were being declared, but the name of the new type is written where the variable
name would be written, and the keyword typedef precedes the declaration.

64. Use of a type name declared by typedef is equivalent to substituting the declaration ex-
pression following the type name into the definition of the type name. I.e., in

typedef int * list_of_ints;
list_of_ints *p;

the second declaration is equivalent to

int * (* p);

i.e., the declaration expression *p from the second declaration is substituted for the type
name list_of_ints in the first declaration.

65. Functions that output values take arguments that are pointers to variables that will hold these
output values. Thus read_list takes pointers for its three arguments, because it will
output three values.

The header file is part of the detailed design of the whole program. After writing it, or rather
a first draft of it (unlike the above final draft), the next questions would be: can one write the
subroutines specified by the header file, and if so, can one write the main program. Its clear in this particular case that in a general sense all this can be done; what is not so clear is that we have all the interface details right. The way to determine whether the interface details are right is to write the main program first. This is called *top down design*, because the main program is the first thing called, the top of the execution path, and it calls the other subroutines, and not vice versa. If we decided instead that it might be impossible to write a list reading subroutine, we would design that first, and would be doing *bottom up design*.

In order to be sure we have all the details right, we need to have a very detailed design of the main subroutine:

```c
/* Binary Search Program Main Function */

** File: bsearchm.c  
** Author: course  
** Version: 1  
** */

#include "bsearch.h"

#include <stdio.h> /* Printf, gets, sscanf. */  
#include <stdlib.h> /* Qsort. */

/* Comparison function required by qsort. */

** Given pointers to two elements in a list_of_ints, returns  
** -1 if the first is less than the second, 0 if the two  
** are equal, and +1 if the first is greater than the second. */

static int list_compare (const void * e1, const void * e2)
{
    return 0;  
}
```
int main ()
{
    /* List of integers to be sorted. */
    list_of_ints list;
    int length;

    /* Count of bad input lines. */
    int error_count;

    /* Buffer to read input line into. Leaves plenty of room for the usual 80 column line. */
    char line [100];
    int number;

    { Print:
        Type in list of numbers to searched, one per line.
        End with empty line.
    }

    read_list (&list, &length, &error_count);

    qsort (list, length, sizeof (*list), &list_compare);

    { Print:
        Type in numbers to be searched for, one per line.
        End with empty line or control-D.
    }
}
while ({not end of input})
{
    { Read a line from the input. }

    { If line is empty, break from loop. }

    { Read number from line. }

    if ({no number could be read})
    {
        { Print an error message including the 
          undecipherable line. }

        { Increment error_count. }
    }
    else
    {
        { Call binary_search with number and list. }

        { Print result of call. }
    }
}

return { 0 if error_count == 0, else 1 }

Notice that in this detailed design we are quite careful to show the exact calling linkage to the 
read_list and qsort functions. We do this to be sure we have the interfacing details correct. 
Note that all the arguments to read_list are output arguments, so for each we need to pass 
a pointer to a variable, which is computed by the & operator, instead of passing the value of the 
variable (this is similar to the output arguments of scanf).

The qsort function illustrates how functions that can handle arbitrary types of data may be 
written. The facts about C needed to understand this function are:

66. The declaration of the qsort function (in stdlib.h) is

    void qsort (void * list,
                size_t length,
                size_t element_size,
                int (*cmp) (const void * e1, const void * e2));
67. In the declaration of qsort, void is used as a type that means “some type that is not known”. Thus list is a pointer to the first element of a vector whose elements are of type unknown to the qsort function.

68. The type size_t is an integer type, typically unsigned int, which is defined by including stdio.h, and which is suitable for sizes.

69. The argument length in the qsort declaration is the number of elements in list.

70. The argument element_size is the value of the sizeof operator applied to the type of the list element, e.g. sizeof (int) for a list of ints. The sizeof operator returns the number of bytes needed to store one element of the given type.

71. The keyword const says that a value cannot be written, but only read. Thus e1 and e2 are pointers to list elements of unknown type that can only be read by the cmp function.

72. If one has a pointer to an element of unknown type in a list, and one has the size of the element type in bytes available, one can add that element size to the pointer to get the pointer to the next element. Example code for doing this is:

```c
int list [2];
void * p0 = list; /* Points at list[0]. */
void * p1 = (char *) p0 + sizeof (int);
/* Points at list[1]. */
printf ("%d %d\n", /* Prints list[0] and */
* (int *) p0, /* list[1]. */
* (int *) p1);
```

73. To convert a C expression from one type to another, the target type in parentheses is written in front of the expression. E.g. (char *) p0 converts p0 to the type char *. Type conversion operators are called casts.

74. Complex types may be represented by declarations that omit the semicolon and the name of the variable being declared. E.g. a pointer to a char variable q would be declared by char * q; so the type is represented as char *, omitting the q and the ;.

75. In order to add a size in bytes to a pointer, one must convert it to a pointer to a char type, since char types take one byte each. Then one can add the size in bytes, to get the new pointer, still of char * type.
76. C will convert a `void *` type to any pointer type, and any pointer type to a `void *` type, without complaining. Other pointer type conversions cause warning messages from the compiler.

Although one would do the detailed design for the `read_list` function after doing the detailed design for `main` and before coding `main`, we will give the code for `main` before the detailed design for `read_list`, so the reader does not have to switch his attention back and forth between these two functions.

```c
/* Binary Search Program Main Function
 **
 ** File: bsearchm.c
 ** Author: course
 ** Version: 1
 **
 */

#include "bsearch.h"
#include <stdio.h> /* Printf, gets, sscanf. */
#include <stdlib.h> /* Qsort. */

/* Comparison function required by qsort.
 **
 ** Given pointers to two elements in a list_of_ints, returns
 ** -1 if the first is less than the second, 0 if the two
 ** are equal, and +1 if the first is greater than the second.
 */
static int list_compare (const void * e1, const void * e2)
{
    return * (int *) e1 < * (int *) e2 ? -1 :
          * (int *) e1 == * (int *) e2 ? 0 :
                   1;
}
```
int main ()
{
    /* List of integers to be sorted. */
    list_of_ints list;
    int length;

    /* Count of bad input lines. */
    int error_count;

    /* Buffer to read input line into. Leaves plenty of room for the usual 80 column line. */
    char line [100];
    int number;

    /* Input and sort list. */
    printf ( "Type in list of numbers to be searched, "
             "one per line.\n"
             "End with empty line.\n"
             "\n" );
    read_list (&list, &length, &error_count);
    qsort (list, length, sizeof (*list), &list_compare);

    printf ( "\n"
             "Type in numbers to be searched for, "
             "one per line.\n"
             "End with empty line or control-D.\n"
             "\n" );
}
while (gets (line))
{
    if ( 1 != sscanf (line, "%d", &number))
    {
        printf ("CANNOT UNDERSTAND %s\n", line);
        ++ error_count;
    }
    else printf
        ( "binary_search (%d, list, %d) = %d\n",
         number,
         length,
         binary_search (number, list, length) );
}

return error_count == 0 ? 0 : 1;
}

The detailed design of read_list is:
/* Function that Reads Lists of Ints
 ** File: readlist.c
 ** Author: course
 ** Version: 1
 **
*/

#include "bsearch.h"
#include <stdio.h> /* Printf, gets, sscanf. */
#include <stdlib.h> /* Malloc. */
void read_list
    ( list_of_ints * list,
      int * length,
      int * error_count )
{
    /* Temporary place to store numbers as they are read,
     ** until we find out how many numbers there are. 
     */
    int temp_list[MAXIMUM_LIST_SIZE];

    { Set length to 0. }
    { Set error count to 0. }

    while ({not end of input})
    {
        { Read a line from the input. }

        { If line is empty, break from loop. }

        { Read number from line. }

        if ({no number could be read})
        {
            { Print an error message including the 
              undecipherable line. }

            { Increment error_count. }
        }
        else if ({list is about to overflow
                  MAXIMUM_LIST_SIZE})
        {
            { Print overflow error message. }

            { Increment error_count. }
        }
    }
A main idea behind this design is that the large list `temp_list` with `MAXIMUM_LIST_SIZE` elements is allocated in the stack, exists only while the `read_list` function is running, and is efficiently allocated and deallocated, as all stack variables are. The final list is allocated in main memory by the `malloc` function, consumes only as much memory as is needed by the actual list, and exists until it is explicitly freed by the `free` function, independently of any function executions.

The facts about the `malloc` and `free` functions are:

77. The declaration of the `malloc` and `free` functions (in `stdlib.h`) are:
    ```c
    void * malloc (size_t size);
    void free (void *p);
    ```

78. The `malloc` function takes a size in bytes and allocates a piece of memory of that size, returning its address.

79. The `free` function frees a block of memory previously allocated by `malloc`, so the memory may be reused by another call to `malloc`.

80. If `malloc` cannot find memory, it returns zero.

81. The identifier `NULL` is defined as zero and is used as a `missing pointer` value, that is, a pointer value that indicates the absence of a properly meaningful pointer value.
The code for the `read_list` function is:

```c
/* Function that Reads Lists of Ints 
** File:        readlist.c 
** Author:     course 
** Version:    1 
** */

#include "bsearch.h"
#include <stdio.h>   /* Printf, gets, sscanf. */
#include <stdlib.h>  /* Malloc. */

void read_list
  ( list_of_ints * list,
    int * length,
    int * error_count 
  )
{
    /* Temporary place to store numbers as they are read,
    ** until we find out how many numbers there are.
    */
    int temp_list[MAXIMUM_LIST_SIZE];

    /* Buffer to read input line into. Leaves 
    ** plenty of room for the usual 80 column line. 
    */
    char line [100];
    int number;

    *length = 0;
    *error_count = 0;
```
/* Read numbers into temp_list. */
while (gets (line) && line[0] != 0)
{
    if ( 1 != sscanf (line, "%d", &number) )
    {
        printf ("CANNOT UNDERSTAND %s\n", line);
        ++ *error_count;
    }
    else if ( *length == MAXIMUM_LIST_SIZE )
    {
        printf ("TOO MANY NUMBERS IN LIST\n");
        ++ *error_count;
    }
    else
    {
        temp_list[(*length)++] = number;
    }
}

/* Allocate list and copy numbers into it */
*list = malloc (*length * sizeof (int));

if (*list == NULL)
{
    /* Memory allocation failure. */
    printf ("ERROR: OUT OF MEMORY\n");
    exit(2);
}
10.5. THE FINAL BINARY SEARCH PROGRAM

```c
{   int * p = temp_list,
    * q = *list,
    * endp = p + *length;

    while (p < endp) *q++ = *p++; }
```

Note that we must put a * in front of every output argument every time we use it.

The `read_list` function uses a few additional facts about C:

82. The ++ operator may be used as either a prefix or postfix operator on a variable. It increments the variable. The value returned by the operator expression is the variable after incrementing for prefix ++, and the variable before incrementing for the postfix ++ operator. Note the postfix operators are performed before the prefix operators (so the parentheses are necessary in (*length)++ above). There is also a similar decrementing operator --.

83. The idiom *p++ for some pointer variable p is common in C. It means to increment the pointer p to point to the next element of a vector, while returning the value of the element pointed at before the pointer was incremented.

84. The `exit` function terminates the program, closing all open files, and returns its single integer argument as the program status, which would normally be the value returned by the main function.

All the above, along with the binary search subroutine from `bsearch3.c`, combines to make a `bsearch` program. The Makefile commands for compiling and testing this program are:

```
bsearchm.o:   bsearch.h bsearchm.c
        gcc -g -ansi -pedantic -c bsearchm.c

readlist.o:   bsearch.h readlist.c
        gcc -g -ansi -pedantic -c readlist.c

bsearch.o:   bsearch.h bsearch.c
        gcc -g -ansi -pedantic -c bsearch.c
```
bsearch: bsearch.h bsearchm.o readlist.o bsearch.o
       gcc -g -ansi -pedantic -o bsearch \
              bsearchm.o readlist.o bsearch.o

bsearch.out: bsearch bsearch.in
             -bsearch <bsearch.in >bsearch.out 2>&1

With an input file bsearch.in that has the same data, list and numbers to be searched for, as
previously, the output is the same as previously. However, as one input line has bad data, the error
count is non-zero, and bsearch returns status 1. To get the make program to ignore this error
indicating status, the command executing bsearch in the Makefile is prefixed by a -.

10.6 The Heap Sort Program

We have covered most of the basics of C, but have not yet dealt with certain fundamental constructs,
such as C structures. Also, we have given no exercises; and its not really possible to learn a
language without using it.

We shall kill both omissions with one exercise. The excercise is to write the hsort program,
that performs heapsorts. We give you the header files, the detailed design files, the Makefile,
and even a library with the object code of all the subroutines. So we actually give you a working
program, but not sources. Your task is to write the sources and test them. You may do this one
subroutine at a time, so the assignment is very modular, and you can go as fast or as slow as you
like.

We will not, however, tell you about C struct and enum types that are used in hsort. For
these you will have to consult your C textbook.

10.6.1 Heap Sort Header Files

There are two header files for the heap sort program. The hsort.h file is for the program itself,
and the hdatum.h file defines the kind of data being sorted, so this may be changed easily.

The hsort.h file begins with an explanation of what a heap is:
/* Heap Sort Program Header
**
** File: hsort.h
** Author: course
** Version: 1
**
*/

/************
****HEAPS:
************
**
** This program stores data in a "heap", and retrieves the data
** from the heap in sorted order.
**
** A heap contains nodes. Each node has a single datum, and two
** pointers at other nodes, called the left and right children.
** There is a single top node, the "root". The pointers can
** take the special value NULL to indicate there is no left or
** right child. Every node is reachable from the root. If the
** data were integers, the following would be an example of a
** heap:
**
** root ---> 5
**     / \ 
**    /   
**   /     
**  /       
** /         n abbreviates NULL
**   / 
** 6    13
**     / 
**    / 
**   n 7   / 
**      /   44 26
**     / 
**    n n / 
**   /   / 
**  n n n n
**
** Such a data structure is called a binary tree. A binary tree is a "heap" if the datum at each node is less than or equal to the datum at each of the node’s children:

```
  d0  d0 <= d1
  / \            
 d1  d2  d0 <= d2
```

** A binary tree is called "balanced" if each node has just as many left descendants as right descendants. A "heap" is supposed to be balanced. Actually, our heaps will not be exactly balanced, and therefore they should be called "partially ordered binary trees" instead of heaps. However, our sorting algorithm has no other possible name than "heap-sort".

** The word "heap" is also used with a totally different and independent meaning, as a place to allocate arbitrary data.

*/

The `hsort.h` file continues by documenting the commands recognized by the `hsort` program (whose code is in `hsort.c`).

/****** HSORT COMMANDS:  ***********/

** The `hsort` program implements a simple command language that permits data to be entered into a heap and be removed from the heap, either one datum at a time, or in batches. The state of the heap may also be printed.

** The type of datum stored in the heap, plus functions that compare and print data of that type, are defined in `hdata.h`.**
** The commands that may be entered at the > prompt are:

**
** put <datum> Put one datum.
**
** get Get one datum.
**
** print Print heap.
**
** input Input many data, one per line.
**
** <datum>  
** <datum>  
** . . .  
** <empty line>
**
** output Output (get) all the data in the heap.
**
** clean Empty the heap without outputting any data.
**
** echo Toggle the echo switch, which echos commands on the standard output.
**
** #<any text> Comment line.
*/

The hsort.h file then defines the major data structures used by the program. The data values to be sorted are of a type hdatum defined separately in the file hdatum.h. This file also defines functions for comparing values, reading (scanning) values, printing values, and deleting values (i.e., freeing any memory that might have been allocated to hold part of the value, such as a character string name).

/********************
**** DATA STRUCTURES:
********************
*/
#include "hdatum.h" /* Type hdatum, functions hd_compare, */
/* hd_scan, hd_print, hd_delete. */
struct node
{
    hdatum value; /* Value stored at this node. */
    int count; /* Number of nodes in the heap rooted at this node. */
    struct node * left; /* Pointer to left child. */
    struct node * right; /* Pointer to right child. */
};

typedef struct node node;

typedef node * heap; /* A heap is a pointer to the root node. */

heap current_heap; /* Our one and only heap. */

The heapsort main program executes its commands by performing operations on the current_heap. These operations are given next.

    /*******************
    **** MAIN FUNCTIONS:
    *******************
    */

    /* Function to initialize the heap to be empty. Actually a macro. */
    #define heap_initialize() (hp_initialize (&current_heap))


/* Function to insert an hdatum v into the heap. Actually a
** macro. */
#define heap_insert(v) (hp_insert (&current_heap, (v)))

/* Function to remove the smallest value from the heap. Returns
** the hdatum removed. MUST NOT be called on empty heap.
** Actually a macro. */
#define heap_remove() (hp_remove (&current_heap))

/* Function to test for an empty heap. Returns true if the
** heap is empty and false otherwise. Actually a macro. */
#define heap_empty() (current_heap == NULL)

/* Function to print the heap. Actually a macro. */
#define heap_print() (hp_print (current_heap, 0))

Lastly, the above operations on the current_heap are actually macros defined in terms of
more general functions that operate on any heap. These functions need to be more general because
they are recursive, and operate on the children of a node, which can be viewed as subheaps.

/************************
**** AUXILIARY FUNCTIONS:
*************************/

/* Prototypes for functions used above, but not seen directly by
** the end user. Needed here so user’s code while have them when
** it calls above macros. */

/* Function to initialize a given heap. Frees nodes of heap and
** leaves heap empty (== NULL). Does nothing when called with
** empty heap. */
void hp_initialize (heap * hp);
/* Function to insert an hdatum into a heap. */
void hp_insert (heap * hp, hdatum number);

/* Function to remove the smallest hdatum from a heap and return it. MUST NOT be called on empty heap. */
hdatum hp_remove (heap * hp);

/* Function to print heap given root and the level of indentation. The level of indentation is the depth we are at in the tree being printed, and is NOT the same as the number of columns to be indented (which is generally some multiple of the depth). The top level is zero. */
void hp_print (heap hp, int level);

As mentioned above, the type of data sorted by our heapsort program is defined separately by the hdata.h file so that this type may be easily changed. Thus our heapsort program is modularized into two modules, one that defines the heapsort program itself, and one that defines the type of data to be sorted. We could have modularized the program into more modules if we wanted; for example, we could have separated the heap data type from the main program.

The hdatum.h file begins by describing the kind of values we will be sorting:
/* Heap Sort Data Definition
**
** File:       hdatum.h
** Author:     course
** Version:    1
**
*/

/**************
**** HEAP DATA:
**************

** This file defines the heap value data type used by the hsort
** program. Value data may be scanned from an string, printed
** to a string, compared, and deleted.
**
** To change the value type used by hsort, only this file need
** be changed.
*/

/* For the current version of this file, an hdatum is a person at
** a college, with a category and a name. Hdata are ranked by
** seniority first and by alphabetical name second.
**
** On input or output, an hdatum has the format:
**
**     <person’s name>, <category>
**
** where
**
**     <category> ::= freshman | sophomore | junior |
**               senior | graduate
**
** A name is separately stored by a call to malloc(), and is
** storage auxiliary to its hdatum value.
*/

The hdatum.h file goes on to define data types.
/* Category of person at a college. This definition is
** private to the hdatum module. The numeric value is in order
** of increasing seniority.
*/
typedef enum {
  HDATUM_NULL = 0, /* Unknown. */
  HDATUM_FRESHMAN = 1,
  HDATUM_SOPHOMORE = 2,
  HDATUM_JUNIOR = 3,
  HDATUM_SENIOR = 4,
  HDATUM_GRADUATE = 5
} hdatum_category;

typedef struct {

  /* The elements defined below are private to the hdatum
   ** module and must not be accessed outside that module.
   */

  hdatum_category category; /* Category of person. */
  char * name; /* Name of person. */

} hdatum;

Lastly, the hdatum.h file defines the functions that will be available for manipulating hdatum values.
/* Function returning -1, 0, +1 according to whether v1 < v2,
** v1 == v2, or v1 > v2.
*/
int hd_compare (hdatum v1, hdatum v2);
/* Function to read hdatum from a string. Updates *pointer to ** point at end of hdatum if success. Allocates any auxiliary ** storage for hdatum if success. Returns true if success ** and false if failure. Returns hdatum in *v on success. */
int hd_scan (char ** pointer, hdatum * v);

/* Function to print hdatum, by copying it into string. ** Updates *pointer to point at end of hdatum representation ** in string. Truncates representation to be at most the ** given number of characters. */
void hd_print (char ** pointer, int number, hdatum v);

/* Function to delete any auxiliary storage associated with ** an hdatum. Hdatum should be discarded after this is done. */
void hd_delete (hdatum v);

10.6.2 Heap Sort Function Detailed Designs

There are four functions defined by hsort.h, a main function defined in hsort.c, and four functions defined by hdatum.h, for a total of nine functions. Here we simply give the detailed design of all nine of these functions without comment. The nine functions have been put into nine separate files so that they may be coded by nine different people (or one person at nine rather different times). For each function we plan on having two files: a .dd file holding the detailed design, which is what we give here, and a .c file holding the code, which is what you are to write. The later file may be made by copying the former file and editing it.

Here are the nine .dd files:
/* Heap Initialization Function Detailed Design
 **
 ** File: hinit.dd
 ** Author: course
 ** Version: 1
 **
 */

#include "hsort.h"

/* Function to initialize a given heap. Frees nodes of heap and
** leaves heap empty (== NULL). Does nothing when called with
** empty heap.
*/
void hp_initialize (heap * hp)
{
    if ( {heap not empty} )
    {
        hp_initialize ( {left node of root of heap} );
        hp_initialize ( {right node of root of heap} );
        free ( {root of heap} );
        * hp = NULL;
    }
}

/* Heap Insert Function Detailed Design
 **
 ** File: hinsert.dd
 ** Author: course
 ** Version: 1
 **
 */

#include "hsort.h"
/* Function to insert an hdatum into a heap. */
void hp_insert (heap * hp, hdatum datum)
{
    if {heap is empty}
    {
        * hp = (node *) malloc {size of node};
        {new node value} = datum;
        {new node left and right child} = NULL;
        {new node count = 1};
    }
    else if ( hd_compare ({root node value}, datum) <= 0 )
    {
        /* Old root value is a minimum. Pick the child
           ** the with fewest nodes to balance the tree. */
        {root count} += 1;
        if ( {left child empty}
             ||
             ({right child not empty}
              &&
              {left child count} <= {right child count})
        )
            hp_insert ({left child}, datum);
        else
            hp_insert ({right child}, datum);
    }
else
{
    /* Old root value is not a minimum. Switch it
    ** with the new datum and recurse with old
    ** root value as the datum.
    */
    hdatum temp = {root node value};
    {root node value} = datum;
    hp_insert (hp, temp);
}

#include "hsort.h"

/* Function to remove the smallest hdatum from a heap and return
** it. MUST NOT be called on empty heap.
*/
hdatum hp_remove (heap * hp)
{
    assert {heap not empty};

    hdatum result = {root datum};

    if {both children are NULL}
    {
        /* Remove the root (and only) node.
        */
        free {root node};
        *hp = NULL;
    }
else if (left child is NULL)
{
    /* Make the right child the root and free
     * the old root.
     */
    *hp = (right child);
    free (root node);
}
else if (right child is NULL)
{
    /* Make the left child the root and free
     * the old root.
     */
    *hp = (left child);
    free (root node);
}
else if (hd_compare
    ((right child's value), (left child's value))
    <=
    0)
{
    /* Right child has new minimum from heap.
     */
    (root datum) = hp_remove (right child);
    --(root count);
}
else
{
    /* Left child has new minimum from heap.
     */
    (root datum) = hp_remove (left child);
    --(root count);
}
return result;
/* Heap Print Function Detailed Design */

/* File: hprint.dd */

/* Author: course */

/* Version: 1 */

/* */

#include "hsort.h"

/* Function to print heap given root and the level of indentation. The level of indentation is the depth we are at in the tree being printed, and is NOT the same as the number of columns to be indented (which is generally some multiple of the depth). The top level is zero. */

void hp_print (heap hp, int level)
{
    printf {5 * level spaces};
    printf [level] followed by spaces to make 5 columns;
    if {hp is empty}
    
        printf {"<empty>\n"};
    
    else
    
    
        hd_print {root value};
        print {carriage return};

        hp_print ( {left child}, level + 1);
        hp_print ( {right child}, level + 1);

}
10.6. THE HEAP SORT PROGRAM

/* Heap Sort Main Program Detailed Design
**
** File: hsort.dd
** Author: course
** Version: 1
**
*/

/* The commands that may be entered at the > prompt are:
*/
char usage [] =
"    put <datum>   Put one datum.\n"
"    get          Get one datum.\n"
"    print        Print heap.\n"
"
"    input        Input many data, one per line.\n"
"    <datum>\n"
"    <datum>\n"
"    . . .\n"
"    <empty line>\n"
"
"    output       Output (get) all the data in the heap.\n"
"    clean        Empty the heap without outputting data.\n"
"    echo         Toggle the echo switch, which echos\n"    commands on the standard output.\n"
"    #<any text>  Comment line.\n"
main ()
{
    /* Buffer into which next command line is read.
    ** Carriage return is stripped from line, which is
    ** ended by a '\0' character.
    */
    char line [100];

    /* Echo switch.
    */
    int echo = 0;

    while ({read next input line into line buffer},
       {end of file not encountered})
    {
        if (echo) {print line read}

        if {line begins with "put"}
        {
            hdatum datum;
            char * p = line + 3;
            if (hd_scan (&p, &datum))
                heap_insert (datum);
            else printf
                    "ERROR: bad datum: {line + 3}\n";
        } else if {line begins with "get"}
        {
            if (heap_empty())
                printf "Heap is empty.";
            else hd_print (heap_remove());
            printf {carriage return};
        } else if {line begins with "print"}
        {
            hprint ();
        }
else if {line begins with "input"}
{
    while (read next input line into line buffer),
        {end of file not encountered})
    {
        if (echo) {print line read}
        if {line empty} break;

        hdatum datum;
        char * p = line;
        if (hd_scan (&p, &datum))
            heap_insert (datum);
        else printf
            "ERROR: bad datum: {line}\n";
    }
}
else if {line begins with "output"}
{
    while (!heap_empty())
    {
        hd_print (heap_remove());
        printf {carriage return};
    }
}
else if {line begins with "clean"}
{
    heap_initialize ();
}
else if {line begins with "echo"}
{
    echo = echo ^ 1;
}
else if {line begins with "#"} /*! Do nothing. */;
else
{
    printf
        ("ERROR: cannot understand line:\n");
    printf {line};
    {print usage};
}
}
return 0;

/* Heap Sort Data Comparison Function Detailed Design
**
** File: hdcomp.dd
** Author: course
** Version: 1
**
*/

#include "hdatum.h"

/* Function returning -1, 0, +1 according to whether v1 < v2, 
** v1 == v2, or v1 > v2.
*/
int hd_compare (hdatum v1, hdatum v2)
{
    if ( {v1 category < v2 category} ) return -1;
    else if ( {v1 category > v2 category} ) return +1;
else
{
    /* Categories are equal. */
{
    Find first character position in names of
    v1 and v2 at which the names differ, or
    if none, find the end (NUL character
    position) of both names.
}
{
    Let c1 and c2 be the characters in the names
    of v1 and v2 at the position found.
}
/* Here we use the fact that the name strings
** end in NUL == 0, and this character is less
** than any other. */
if (c1 < c2) return -1;
else if (c1 > c2) return +1;
else if (c1 == c2) return 0;
}

/* Heap Sort Data Read Function Detailed Design
**
** File:       hdscan.dd
** Author:    course
** Version:   1
**
*/
#include "hdatum.h"
/ Function to read hdatum from a string. Updates *pointer to
** point at end of hdatum if success. Allocates any auxiliary
** storage for hdatum if success. Returns true if success
** and false if failure. Returns hdatum in *v on success.
*/

int hd_scan (char ** pointer, hdatum * v)
{
    /* Scan with a temporary copy of *pointer in case we
     ** decide to give up later. */
    char * p = * pointer; /* Next input character. */

    /* Temporary to hold name. */
    char name[101];
    char * np = name; /* Next name character to use. */

    { scan over whitespace }
    { scan till ',', or end of string, copying characters
to name; truncate name if too many characters }
    { put '\0' at end of name }
    if ( {input not at comma} ) return 0;
    { scan over whitespace }
    if ( {input equals "freshman"} )
        v.category = HDATUM_FRESHMAN;
    else if { etc. through "graduate" }
    else return 0;
v.name = (char *) malloc ( {length of name} );

/* This program does not expect to run out of memory and is not very smart about doing so. */
assert (v.name != NULL);

{ copy name to v.name }
* pointer = p;

return 1;
/* Heap Sort Data Deletion Function Detailed Design
**
** File: hddelete.dd
** Author: course
** Version: 1
**
*/

#include "hdatum.h"

/* Function to delete any auxiliary storage associated with
** an hdatum. Hdatum should be discarded after this is done.
*/
void hd_delete (hdatum v)
{
    if (v.name != NULL) free (v.name);
}

10.6.3 Heap Sort Program Makefile

The Makefile for the heapsort program exercise contains the lines:
OBJECTS =
    # As you write the corresponding .c file, add the following
    # to OBJECTS:
    # hdcomp.o
    # hddelete.o
    # hdprint.o
    # hdsan.c
    # hinit.o
    # hintert.o
    # hprint.o
    # hremove.o
    # hsort.o

and
# Because objects are listed first when given to gcc, they take precedence over functions of the same name in hsort.a
hsort: $(OBJECTS) hsort.a
gcc -g -ansi -pedantic -o hsort $(OBJECTS) hsort.a

hsort.out: hsort.in hsort
    -hsort <hsort.in >hsort.out 2>&1

If you just run make without doing anything else, the hsort program will be linked using versions of the nine functions stored in the library hsort.a. If you write two files named, for example, hinsert.c and hprint.c, and change the definition of OBJECTS to:

    OBJECTS = hinsert.o hprint.o

then make and gcc will use your two functions instead of the code stored in hsort.a to link hsort.

By this means you can successively write and test your own versions of the nine functions that comprise the heapsort program.
Chapter 11

C++

by Jon McAuliffe and Mark Immel

11.1 Introduction

11.1.1 What is C++?

C++ is a programming language developed by Bjarne Stroustrup and his associates at AT&T in the early 1980’s. Its features are meant to serve as an enhancement to and extension of the enormously popular and widely used C programming language; as such, the functionality of C remains as a subset of C++ (with a few small exceptions discussed later). Readers familiar with the ++ increment operator in C will note that the language’s name reflects this intent.

The language facilities provided by C++ which are absent in C can be divided into two general categories: those which enforce and refine techniques already a part of the procedural programming paradigm realized by C, and those which support software development using the object-oriented programming paradigm. Object orientation is an evocative but often poorly understood catch-phrase of modern-day programming, so in this introduction we shall turn our attention briefly to defining it and sketching the ways C++ supports it. To begin, however, we outline the first type of extension present in C++. Subsequent sections will provide a comprehensive discussion of all the features relevant to CS51.

11.1.2 Non-Object-Oriented Features of C++

A detailed account of these features follows this introduction, but it is useful to characterize the motivations for the non-object-oriented addenda to C supported by C++. One apparent goal is to make C++ not just a strongly typed language, like C, but also a strongly typechecked language.
For example, the \texttt{#define} preprocessor directive is widely used in C programs to create named constants and macros, yet it inherently confounds typechecking for reasons discussed in Section 2. Because of this, C++ tries to eliminate the use of \texttt{#define} by providing other facilities in the language to handle constants and macros (see Use and Scope of Symbolic Constants and Function Inlining below for details).

C++ also places major emphasis on streamlining and simplifying the language interface to very frequent operations. As regards memory management, for example, the \texttt{malloc()} and \texttt{free()} C library routines, which provide a rather clumsy interface to dynamic memory allocation, are replaced by the elegant operators \texttt{new} and \texttt{delete} (see Memory Management: The new and delete Operators below). Further, variables need no longer be defined at the beginning of a block (as they must in C), allowing them to be introduced wherever it is semantically appropriate (see Position of Variable Definitions below). The use of argument passing by reference is greatly simplified with the introduction of a reference variable type (see The Reference Data Type below). And very importantly, both built-in and user-defined functions can be multiply overloaded, allowing the same logical name to be given to similar operations on different variable types (see Function Overloading and Operator Overloading with $<$ and $>$ below).

These topics and more are covered by way of numerous examples in the following chapters, but for now bear in mind that they are all provided with the primary intentions of making programs more type-safe and making code more concise, easier to write, and less complex.

\section{Object Orientation in C++}

A paradigm is, according to Webster’s Dictionary, “an example serving as a model.” When we talk about a \textit{programming paradigm}, we mean a unified approach to solving problems computationally, a \textit{way of thinking} about how to represent our methods for answering questions in some problem domain. Until rather recently, the uncontested favorite way of thinking about solving problems with computers has been the \textit{procedural programming paradigm}, with which every CS51 student is already familiar. The term “functional decomposition” is more widespread and means about the same thing — we choose data structures to represent the parts of the problem we want to solve, then we write a group of functions which collectively solve the problem (usually by calling each other). How else would we approach problem solving with a language like C or Pascal?

In fact, the bottom line answer to this question is that we wouldn’t approach it any other way at all! The reasoning behind object orientation is not that designing data structures and algorithms is useless or old-fashioned. Instead of asking whether the basic approach of procedural programming is wrong, object orientation questions whether data structures and functions are \textit{enough abstraction} to do good programming. What do we mean by “abstraction” in this context? Recall that a variable assignment like $x = 10$ in C is an abstraction for a sequence of machine instructions which carry out that operation (putting the value 10 in the memory location called $x$). As another
11.1. INTRODUCTION

example, a for loop is an abstraction for a bunch of machine instructions which iterate over the loop’s body some number of times. In fact, function calls are an abstraction for machine instructions which switch a program’s execution to some different series of instructions, then return to the old point of execution. So the farthest that abstraction goes in C is allowing us to define functions as well as some separate data structures on which the functions operate. What other kinds of abstraction might we want to use, and why? In the next two sections we briefly motivate two kinds of abstraction which are central to the notion of object-oriented programming: data abstraction and inheritance hierarchies. Both are layers built on top of the functional abstraction used in procedural programming. We introduce only the indispensable terminology of object-orientation as we go, so that the reader is comfortable with a few key ideas used in most object-oriented languages. Since C++ supports both data abstraction and inheritance in some fashion, we leave these details of the language to later.

11.1.4 Data Abstraction

Above we noted that functions and data structures are the highest level of abstraction in C. And yet even though the language only provides abstraction to this level, as C programmers we routinely introduce another “half layer” of abstraction on top of this — namely, the organization of our program’s code into separate files. Why do we use .c and .h files? Because in large programs, it is enormously important to group together in the same place functions that handle one job, and to put the data structures they rely on in a closely related place. Not just good style but our sanity depends, for example, on sticking input/output functions in io.c and structures they use in io.h. If we mixed those functions into other files, they might accidentally change variables which are local to those files because of a naming coincidence, or they might have access to static functions we wanted to remain local to one file, or we might simply not be able to remember where we put them! If we don’t put the input/output functions’ data structures in io.h, we might later forget where they are, or change them accidentally because we didn’t realize which functions they were meant to be used by.

Clearly, the use of .c and .h files is a reasonable programming practice. Unfortunately, this convention has its drawbacks, too. For instance, files are not a logical component of a programming language, they are a logical component of a file system. We might rely on the idea of a file to organize our program, but wouldn’t it be reasonable for the language itself to provide a mechanism for grouping together a set of similar functions, along with the data they operate on? What about in Pascal, where we are only allowed one file — shouldn’t we still have some way of organizing our functions and data, analogous to the way we use .c and .h files? If the answer to this question seems to be yes, then we are already proponents of data abstraction. For the idea behind data abstraction is just to provide some construct in the programming language to group together a set of functions which handle the same job, along with the data they use to do their work. In C++ and
almost everywhere else, such a grouping is called a class.

Another advantage to defining classes, instead of just defining functions and data structures, is reflected by an alternative name for data abstraction, data hiding. If we want many parts of a program to manipulate the same data structure, we make ourselves vulnerable to big problems if we later decide to change that data structure’s definition. When the representation of data changes, then every function which used the old version of the data must also change — that means a lot of functions for big programs. So here is an alternative: put the data structure inside a class, and then add some functions to the class which provide an interface to the data (just as we might write a bunch of functions which provide an interface to matrix calculations). Now other functions in the program don’t directly use the data, they only ask the interface to manipulate the data for them — the actual representation has been abstracted away. As an example, consider a class which provided a matrix interface:

```cpp
class Matrix {
    int theMatrix[50][50]; // the internal representation
    // of the matrix
    int GetValue(int x, int y); // find a value at a certain
    // location
    int SetValue(int x, int y, int value);
    // set a value at a certain
    // location
    int Determinant(); // calculate the determinant
    // of the matrix
}
```

Notice that the data and the functions are all together inside the class definition. Now other functions in the program can use variables of type Matrix. As long as those other functions never directly use the variable theMatrix, but instead call the interface functions provided, we are completely free to use some other kind of data structure to represent the matrix (for example, a sparse-array representation with linked lists). All we have to do is change the definitions of GetValue(), SetValue(), and Determinant(); the parts of the program which called those interface functions before still call them in the same way.

Now for some terminology. We know already what a class is. When we actually create a variable which has some class as its type, that variable is called an object. (Programs which rely on objects as their primary unit of data are said to be object-oriented.) Since many objects can be created from a certain class (just as many variables can be created of type int), we say that each object is an instance of its class. The representation variables of a class definition are called the data members in C++. The other half of a class definition, the interface functions which manipulate the data members, are called the member functions. Calling a member function of some object
is known as sending a *message* to that object. So at this point we can think of object-oriented programming as designing classes whose instances (objects) interact by passing messages to each other. Since C++ differs in its terminology somewhat from the “classical” usage stemming from SmallTalk, one of the very first object-oriented languages, a summary of the differences is given below.

<table>
<thead>
<tr>
<th>SMALLTALK</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance variable</td>
<td>data member</td>
</tr>
<tr>
<td>method</td>
<td>member function</td>
</tr>
<tr>
<td>send message</td>
<td>call member function</td>
</tr>
</tbody>
</table>

### 11.1.5 Class Inheritance

Data abstraction is a powerful tool for software engineering, and many programming languages provided some form of data abstraction long before “object orientation” became the darling phrase of the computer industry. Nonetheless, data abstraction does not provide us with a mechanism for expressing other important facts about programs which use classes. Suppose we want to write an airport simulation. Since passenger planes and military planes fly at different speeds and altitudes, must land in different parts of the airport, and so on, we would probably define both a *PassengerPlane* class and a *MilitaryPlane* class. Yet obviously much of both classes would be the same — both kinds of planes have some amount of fuel left in them, both need to put down landing gear when they make an approach, etc:

```cpp
class PassengerPlane {
    int gallonsLeft;
    // remaining fuel
    int numPassengers;
    // how many passengers on board
    //...
    void LandingGear();
    // put landing gear up or down
    int ReportFuel();
    // return the amount of fuel remaining
    void UnloadPassengers();
    // thanks for flying. have a nice day
}
```
class MilitaryPlane {
    int gallonsLeft;
    // remaining fuel, just like passenger plane
    int cargoWeight;
    // no passengers, but x lbs. cargo instead
    //...
    void LandingGear();
    // put landing gear up or down
    int ReportFuel();
    // return the amount of fuel remaining
    void UnloadCargo();
    // sign for these, sir
}

It seems like a rather intuitive idea to “factor” these similarities out of both airplane classes, if only to save ourselves the trouble of writing exactly the same code twice in two classes! And what if we add private planes — they can’t fly as high or as fast and they run out of fuel more quickly, but they still have some information about them in common with passenger jets and military aircraft. Given the current state of our tools for using data abstraction, we are faced with the prospect of duplicating a lot of code every time we think of a new kind of airplane. A nice solution to the problem would be to define a plain old Airplane class, which contained, say, gallonsLeft, LandingGear(), and ReportFuel(). If we could then somehow indicate that PassengerPlane and MilitaryPlane each contain all the information and behavior of the class Airplane, our only further work would be to define the data members and member functions specific to passenger and military planes. In other words, using some kind of class inheritance mechanism would save us the trouble of rewriting all the shared state and behavior among different kinds of airplanes. The result might look something like this:

    class Airplane {
        int gallonsLeft;
        // all planes have some amount of fuel
        void LandingGear();
        // all planes have landing gear, we hope!
        int ReportFuel();
        // all planes must be able to check fuel
    }
class PassengerPlane "inherits-from" Airplane {
    // We can imagine that the data members and
    // functions of class Airplane are implicitly
    // written right here, before the code below!

    int numPassengers;
    // how many passengers on board
    //...
    void UnloadPassengers();
    // thanks for flying. have a nice day
}

class MilitaryPlane "inherits-from" Airplane {
    // We can imagine that the data members and
    // functions of class Airplane are implicitly
    // written right here, before the code below!

    int cargoWeight;
    // no passengers, but x lbs. cargo
    // instead
    //...
    void UnloadCargo();
    // sign for these, sir
}

Now PassengerPlane and MilitaryPlane contain only the variables and functions which make them differ from a generic airplane. If we call the LandingGear() function on an object of the PassengerPlane class, the passenger plane knows to use the function it inherited from its parent class Airplane. Actually, we have a tremendous amount of flexibility in deciding how inheritance works. There can be more than two levels of inheritance, so we might define a special kind of passenger plane which inherits from the PassengerPlane class (and therefore, transitively, also from the Airplane class). Children classes can override inherited data and functions if they want, implementing special versions of generic inherited behavior. It is even possible to give a single class more than one parent, but this multiple inheritance is not something we will touch on in CS51. In fact, using class inheritance buys us a lot of other advantages in terms of reusability and reliability of our programs, but we don’t want to examine this issue any further here.
(Of course "inherits-from" is not actually the correct C++ syntax for use in class definitions. To be correct, replace it with "public". But we get ahead of ourselves.)

Again, we mention just a little terminology which is prevalent in the world of object-oriented programming. In C++, if `PassengerPlane` is defined to inherit from `Airplane`, then `Airplane` is a base class and `PassengerPlane` is a derived class. Every class hierarchy or "inheritance tree" has a topmost class which doesn’t inherit from anything; this class is called the root class (it’s the “root” of the tree). Sometimes we define a class whose only purpose in existence is to serve as the basis for inheritance; that is, we would never create objects of that class in a program, only objects of its derived classes. Such a class is known as an abstract class precisely because it is used only to abstract some common state and behavior out of classes which will then inherit from it. As with data abstraction, C++ has veered away from standard usage in its inheritance terminology, so we once more provide a table, below, showing how C++ terms compare to terms from the usage stemming from SmallTalk.

<table>
<thead>
<tr>
<th>SMALLTALK</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>superclass</td>
<td>base class</td>
</tr>
<tr>
<td>subclass</td>
<td>derived class</td>
</tr>
<tr>
<td>abstract superclass</td>
<td>abstract class</td>
</tr>
</tbody>
</table>

### 11.2 Non-Object-Oriented Features of C++

We have already mentioned that most valid C programs are also valid C++ programs. The first part of this section discusses the exceptions to this rule, the parts of the C++ language which introduce incompatibilities with ANSI C (the strict specification of C). Some of these features are simply questions of good style, and are already incorporated into most modern-day C compilers. They will therefore be familiar to the reader. Many, however, reflect essential incompatibilities between C and C++, and would have to be changed if a program written in C were given to a C++ compiler.

The second part of the section deals with features which are brand-new to C++, rather than conflicting implementations of ideas from C. The extent of their usefulness ranges from obvious to very subtle, so we will try to demonstrate their appropriate application by way of examples.

Finally, in the third part of this section we provide extremely brief coverage of the input/output facilities supplied by the C++ iostream library. The sheer quantity of functionality provided by the package prevents us from even outlining it in its entirety.

Some non-object-oriented features of C++ are omitted here either because they are irrelevant to CS51 or because they are not yet supported by some of the C++ compilers used in the course. While it should not prove necessary to investigate such features, most are discussed in the recommended text.
11.2. NON-OBJECT-ORIENTED FEATURES OF C++

11.2.1 Non-Object-Oriented Features of C++ that Conflict with ANSI C

11.2.1.1 Inclusion of C Library Headers in C++ Files

Whenever a program calls a function from a predefined library, the library header file must be included using the `#include` preprocessor directive, so that the compiler obtains the information it needs about the function. This is true in C and remains true in C++. However, when including header files for standard C files in a C++ program, a special syntax is needed:

```cpp
extern "C" { #include <stdio.h> } // for printf(), scanf(), etc.
```

Multiple `#include` s can occur between the curly braces, separated by semicolons. A C++ library header, of course, is included in the normal way:

```cpp
#include <iostream.h>
// includes declarations for C++ iostream package
```

Inclusion of C library headers requires a special syntax because C++, by default, changes (“mangles”) the name of every declared function, for reasons we will discuss later.

11.2.1.2 Definition of Aggregate Data Types

To define a new data structure which is a collection of other data types in C, we would write something like

```c
struct employee {
    int id_num;
    char *lname;
    float salary;
};
```

and to create variables of this type, we would have to write a definition like

```c
struct employee best_employee;
```

Since it is cumbersome to use `struct employee` in the definition of every variable of this type, C programmers usually enclose the `struct` in a `typedef`.

```c
typedef struct _employee {
    int id_num;
    char *lname;
    float salary;
} employee;
```

so that variables of this type can be more succinctly defined, like so:

```c
employee best_employee;
```
In C++, it is permitted but not necessary to use the `typedef` for abbreviatory purposes: we could make the above variable definition immediately after defining the `struct`, though it does no harm to use the `typedef` as well. In this respect C++ differs from C. However, since the `typedef` is necessary in a C program, it never hurts to use it in any code compatible enough with C to be handled by a C compiler. Note, by the way, that the above discussion of `struct` holds for the other type aggregates `enum` and `union` as well.

### 11.2.1.3 Naming Conflicts between Variables and Data Types

Continuing with the example above, it would be possible for us to define a variable of type `employee` which use the identifier `employee` as its name:

```c
employee employee; /* this is very bad style, but works in C */
```

Although such a definition leads to a great deal of confusion and is extremely poor style, a C compiler will not complain about it. However, in C++, there is no distinction made between names used for data types and names used for variables — in other words, they share the same `namespace`. So the above definition would cause a compile-time error if given to a C++ compiler. We hope this will not break any hearts, as no conscientious programmer ought to use data type names as variable names anyway.

### 11.2.1.4 Use and Scope of Symbolic Constants

Named constants are a very important idea, both in C and C++. They allow the programmer to replace obscure references to “magic numbers”, as below, with meaningful names, greatly improving understandability.

```c
#define KM_PER_MILE 1.6 // roughly

// first line below is poor style - 1.6 is a "magic number"
float howManyMiles = DoConversion(miles, 1.6);

// MUCH better
float howManyMore  = DoConversion(otherMiles, KM_PER_MILE);
```

In addition, it allows us to change every occurrence of the constant value in the program just by altering the definition of the constant. However, the use of `#define`, a preprocessor directive, means that the named constant is replaced by its defined value everywhere in the program before the compiler sees the code. This has two big drawbacks: the constant can’t have a type, since typechecking is done during compilation, after the constant has been replaced with its value; and debuggers have no information about named constants, making it that much more difficult to fix an
ailings program. C++ tries to eliminate these problems by providing better facilities for constants in the compiler, making the `#define` directive unnecessary. In C++ we define a constant as follows:

```cpp
// now compiler can typecheck constant...
const float KM_PER_MILE = 1.6; // value can NEVER change
// ...and debugger knows about it
float howManyMiles = DoConversion(miles, KM_PER_MILE);
```

Note that we must give the constant an initial value when we define it, since it is subsequently never allowed to be on the left-hand side of an assignment. A constant of this sort, which is handled by the compiler and not the preprocessor, is known as a *symbolic constant*. While symbolic constants are available in C, they are seldom seen because they cannot be used as dimensions of arrays. Such a use is possible in C++:

```cpp
const int HASH_TABLE_SIZE = 101; // remember, we MUST initialize
int mainHashTable[HASH_TABLE_SIZE]; // error in C, allowed in C++
```

Since symbolic constants are meant to be used in place of `#define` constants in C++, they should be put in `.h` header files and not in `.c` files (where they are found in C). One implication of this is that symbolic constants have file scope in C++ but program scope in C. This means that a symbolic constant defined in one `.c` file is automatically visible to every function in every file of a C program. Do not worry too much about this distinction, however. If a `.c` file needs to refer to some constants, just `#include` the header files where the constants are defined, and the scoping issue will then resolve itself. We mention it only because C programs with symbolic constants might, when given to a C++ compiler, fail to link, precisely because the above header file changes and inclusions need to be made.

### 11.2.2 Non-Object-Oriented Features of C++ that Extend ANSI C

#### 11.2.2.1 The `//` comment delimiter

In C++, a comment may either be put between the traditional `/* */` delimiters, or may be delimited by a double-slash `//` and a newline character. In fact, many modern C compilers support this kind of comment as well.

While seemingly harmless, the `//` comment can introduce one very nasty bug when it is used to comment a `#define`. Some older C++ compilers don’t notice a `//` comment on the same line as a `#define` statement, and they therefore *add* it as part of the constant’s defined value:

```cpp
#define KM_PER_MILE 1.6 // CAUTION: this may cause a nasty bug
```

Then instead of substituting `1.6` for `KM_PER_MILE` in the program, the preprocessor substitutes `1.6` and the comment which is on the same line. However, this is not a problem in most C++
compilers, and since we are striving to remove `#defines` from our C++ programs anyway, we ought not to run into any such difficulties.

### 11.2.2.2 Position of Variable Definitions

A **block** in C or C++ is essentially any matching pair of curly braces and everything in between them. C requires us to put all the variable definitions in a block at the block’s beginning.

```c
int AnotherExampleFunction(char *string)
{
    // this brace begins a block
    double someVar = 12.2; // 12.2 should probably be a constant!
    int i, result; // variables are at block's beginning

    DoCalculation(string, someVar);

    for (i = 0; i < someVar; i++)
    {
        // a second block

        result = DoAnything();

    } // end of for-loop's block

    return result;

} // this brace ends the block that defines the function
```

C++ allows variable definitions to come anywhere within a block, provided, of course, that a variable is defined before it is used! So we could rewrite the above function as follows:
int AnotherExampleFunction(char *string)
{
   double someVar = 12.2; // 12.2 should probably be a constant!
   DoCalculation(string, someVar);

   int result; // result variable defined AFTER a function call
   for (int i = 0; i < someVar; i++)
   {
      // a second block
      // i is defined INSIDE the for loop’s parentheses
      // but OUTSIDE the loop’s block braces
      result = DoAnything();
   }
   return result;
} // this brace ends the block that defines the function

This may seem at first like an unimportant feature of the language, but it turns out that putting variable definitions where they are logically appropriate, instead of all lumped together at the beginning of a block, can make code quite a bit clearer.

11.2.2.3 Memory Management: The new and delete operators

Programs that don’t allocate memory until absolutely necessary are well-written programs. Whenever it is not clear at compile-time whether memory is going to be needed for some particular use, for instance as a character buffer, a good programmer will write code that allocates that memory dynamically, and only in those instances where it will actually be used. Such a programmer will, of course, never fail to free that memory once it is no longer needed (it should be clear that this kind of programmer doesn’t actually exist!). C provides for dynamic memory allocation and release via the functions malloc(), for Memory ALLOCation, and free(). Anyone with experience in programming C knows how cumbersome calls to malloc() are — the argument is typically the product of some desired number of “slots” and the size of each slot, and the pointer returned must always be cast to the right type. Also, considering how crucially important dynamic memory allo-
cation is to good programming, it is surprising that arguments to malloc() and free() must be given in parentheses, as if they were functions in some obscure library instead of an essential part of the C programming language. Consider a function which creates a buffer when needed:

```c
void GrabSomeMemory(bool wantMemory, int howManyChars) {
    char *charBuf = NULL; // initialize
    if (wantMemory == TRUE) {
        // dynamically allocate a character buffer
        char *result = (char *)malloc(howManyChars * sizeof(char));
        // hard to read ............................................................
    }
    //...
    if (charBuf != NULL) free(charBuf);
    // don’t forget to free dynamic memory!
}
```

To rewrite this function in C++ (and with luck improve readability), we replace the calls to malloc() and free() with use of the new and delete operators:

```c++
void GrabSomeMemory(bool wantMemory, int howManyCharacters) {
    char *charBuf = NULL; // initialize
    if (wantMemory == TRUE) {
        // dynamically allocate a character buffer
        // use the new operator
        char *result = new char[howManyChars];
    }
    //...
    delete charBuf;
    // don’t forget to free that memory!
}
```

Notice that the syntax for a dynamically allocated array is just like that used to define statically allocated arrays (for which the compiler creates memory at compile-time), with the exception of the keyword new in front as well as the fact that the dimension can be any computable expression, and not just a compile time constant. Also, instead of parentheses around the argument to delete, we simply write the pointer’s name right after the operator — delete handles NULL pointers gracefully, simply doing nothing instead of generating an error.

Similarly, if we define a data type for which we want to create variables dynamically, instead of using sizeof() to tell malloc() how big the structure is, we simply write the data type’s
name after the new operator:

```c
typedef struct { // a really important aggregate type
    int    importantNumber;
    char **strangePointer;
    double bigDecimalValue;
} CrucialDataType;
// ... 
if (needCrucial == TRUE) { // we want some dynamic memory 
    CrucialDataType *importantPointer = new CrucialDataType;
} 
// ... 
delete importantPointer; // free dynamically allocated memory
```

So nothing has changed about the idea of dynamically allocated storage, but the interface to dynamic memory is considerably nicer in C++.

### 11.2.2.4 Function Inlining

In the section on *Use and Scope of Symbolic Constants* above we discussed some reasons why we should try to eliminate the use of the `#define` preprocessor directives from our programs. We talked there about named constants; here we turn to the other use of `#define`, macro creation. *Macros* in C are tiny functions (usually 1-3 lines) whose definitions are never separately compiled. Instead of generating object code for macros, the compiler simply substitutes the entire macro definition in wherever a call to it appears. A common example is a macro for returning the minimum of two values:

```c
#define min(X,Y) (X < Y ? X : Y)
// uses ternary ?: operator
int aNumber = min(3, 20);
```

which eventually arrives at the compiler, after preprocessing, as

```c
int aNumber = (3 < 20 ? 3 : 20);
```

in which the call to `min` has been *macro-expanded*.

One reason we define `min` as a macro is that we would require more time to make the call to a function `min()` and then return from the call than we require to execute the algorithm! Avoiding function call overhead, where appropriate, is still seen as a good idea by C++ programmers, of course, but the main drawbacks to preprocessor directives have not gone away: no typechecking can be done by the compiler, and debuggers never see calls to macros. C++ again addresses this problem by providing a facility to replace the `#define` directive’s behavior, namely the definition of inline functions. We would write the above example in C++ as
inline int min(int x, int y) { return x < y ? x : y; }

Two important points should be made. First, the `inline` keyword is only a recommendation to the compiler — a given function may or may not be inlined, depending on how long it is, whether it calls itself recursively, etc. Second, note that the `#define`'d `min` macro did not have any types for its arguments. This is a double-edged sword. On the one hand we supposedly want to typecheck our programs strongly, yet on the other we are happy (in the case of `min`) that we only need to write a single macro to handle `int`s, `float`s, `double`s, or any other data type accepted by the less-than operator. In C++ it seems, we would have to do something like

```cpp
inline int min(int x, int y) { return x < y ? x : y; }
inline float min(float x, float y) { return x < y ? x : y; }
inline char min(char x, char y) { return x < y ? x : y; }
...
```

In fact, this blatant shortcoming of inline functions is addressed by yet another language extension, templates, but since some of our C++ compilers in CS51 do not support them, for the time being we will laboriously write out every case where necessary. As a final word on inlined functions, be sure to put them in `.h` header files, not `.c` files, since they are meant to replace `#define`'d macros (just as symbolic constants go in header files to replace `#define`'d constants).

### 11.2.2.5 Function Overloading

Suppose we wanted to write some code which lets us print out a number on the screen, whether the number is represented as an `int`, a `double` or a string of `char`s representing digits. Since the operation is logically equivalent, and only the argument types differ, a C programmer might write some functions like this:

```cpp
void DisplayNum_int(int x) {
    printf("%d", x);
}
void DisplayNum_string(char *str) {
    printf("%s", str);
}
// and so on
```

But then, of course, the programmer would need to remember the way in which each function name differs, and be sure to use the correct function for each different argument type he might have. It would be much more convenient to use a single function name, `DisplayNum`, for all the functions, than have the compiler choose the right version based on what arguments are given to a particular invocation. Such an approach to naming similar functions is called function overloading, or sometimes operator overloading, and it is allowed without any special syntax in C++ — we just
11.2. NON-OBJECT-ORIENTED FEATURES OF C++

do it:

```c++
void DisplayNum(int x) {
    printf("%d", x);
}
void DisplayNum(char *str) {
    printf("%s", str);
}
// and so on, ...
```

```c++
int myFavoriteInt = 37;
char *hugeNumber = "25238541209897143578982";
DisplayNum(myFavoriteInt); // compiler picks first definition
DisplayNum(hugeNumber);   // compiler picks second definition
```

Note that the parameters of like-named functions may differ in any way — they can have different types and there may be a different number of them from function to function. The C++ compiler distinguishes functions with the same name by adding some information about argument type and number to the end of each function name. So in our example the final function names might actually turn out something like `DisplayNum__1i` and `DisplayNum__1cp` (or something similar). In fact, C++ actually does this “name mangling” to every function name in a program, whether or not it is overloaded. That’s why we need a special syntax for `#include`’ing ANSI C header files — to keep C++ from mangling the names of C functions used in a program.

### 11.2.2.6 The Reference Data Type

Recall that arguments given in a function invocation cannot be directly modified by the called function, because the argument values are copied into the function’s unique local parameters. If we want a function to be able to modify arguments, rather than just manipulate copies of their values, we use the trick of giving the called function pointers to the arguments:

```c++
void dumb_swap(int arg1, int arg2) {
    int temp = arg1;
    arg1 = arg2;  // this has NO EFFECT on the arguments given
    // to the function
    arg2 = temp;  // NEITHER does this
}
```
void swap(int *arg1, int *arg2) {
    int temp = *arg1;
    *arg1 = *arg2;
    *arg2 = *temp;
}

main() {
    int a = 7, b = 12;       // initial values
    dumb_swap(a, b);
    printf("%d %d\n", a, b); // a still 7, b still 12,
                            // prints out "7 12"

    swap(&a, &b);            // use address-of operator '&
                            // to pass ptrs
    printf("%d %d\n", a, b); // now a == 12, b == 7,
                            // prints out "12 7"
}

This technique, known as *pass-by-reference* function invocation, works correctly, but carries with it the danger inherent to all overcomplex code: things go wrong very fast if the programmer drops a single & or *. To combat this risk, C++ introduces a new data type, the *reference* type. A reference variable, as a function parameter, is best thought of as a particular kind of pointer: one which is automatically assigned to the address of arguments, and automatically dereferenced every time it is used in code. Reworking our example above, we would have
// To define a variable of type "int reference", we use the
// syntax "int &". This is appropriately reminiscent of "int *",
// since references are very similar to pointers.
//
void swap(int &arg1, int &arg2) {
    // The type of the variables passed to swap() should be ints;
    // the above parameters will AUTOMATICALLY be assigned the
    // addresses of the arguments ( e.g. use swap(a, b),
    // NOT swap(&a, &b) ).
    //
    int temp = arg1; // arg1 is AUTOMATICALLY dereferenced
    arg1 = arg2; // both arguments automatically dereferenced
    arg2 = temp; // arg2 automatically dereferenced
}

main() {
    int a = 7, b = 12; // initial values
dumb_swap(a, b); // still has no effect on a or b
    printf("%d %d\n", a, b); // a still 7, b still 12,
    // prints out "7 12"

    swap(a, b); // DON'T use address-of operator
    // '& to pass ptrs
    printf("%d %d\n", a, b); // now a == 12, b == 7,
    // prints out "12 7"
}

As with the new and delete operators, C++ has tried to improve upon a clumsy or complicated interface to an important facility by streamlining syntax. References can be slightly confusing at first, since there is so much implicit dereferencing and address assignment going on, so do not feel obligated to use reference variable types if they cause any unease. The old technique from C of passing by reference still works just as well.

11.2.2.7 Default Function Arguments

Sometimes we use a certain function dozens of times with the same argument values, only giving it different values in rare instances. In such cases it would be convenient for the language to provide a facility which gives a function’s parameters some default values automatically. C++ allows such
default function arguments. If a function’s definition specifies one or more default values, and in some invocation not enough arguments are given to cover all the function’s parameters, then the default values are used:

```cpp
// Specify the default values with "pseudo-assignment" in the
// formal parameter list of the function definition.
float RaiseToPower(int value = 1, int exponent = 2) {
    // return value^exponent
}
```

```cpp
// ...
float answer1 = RaiseToPower(2, 4); // returns 2^4, = 16
float answer2 = RaiseToPower(3);  // returns 3 squared, = 9
float answer3 = RaiseToPower();   // returns 1^2, = 1
```

One limitation seems apparent — default arguments are “prejudiced” towards the right, in the sense that we can’t use a default value for some parameter on the left and then give explicit values for variables further right. In other words, the effect we might try to elicit by writing `RaiseToPower( ,5)`, thereby leaving one argument slot open, is not currently supported by C++. Another difficulty also presents itself in distinguishing the use of default arguments from the use of overloaded functions. How do we know `RaiseToPower` doesn’t actually have three separate definitions, based on how many arguments it is given? What if we want to give default arguments to overloaded functions? The C++ compiler does use a consistent approach to answering such questions, but too many unfathomable subtleties crop up to cover here. The situation is even more grim when we consider that the compiler can automatically promote an `int` to a `float` and so on. In most cases these issues affect our programming tasks only tangentially, so we had best leave them to the language lawyers.

### 11.2.3 Input and Output with C++

Almost every C program makes use of the I/O facilities provided by `<stdio.h>`, the ANSI C standard I/O library. Functions such as `printf()` and `scanf()` typically appear in the very first programs C neophytes learn how to write. Since the names and idiosyncrasies of functions in `<stdio.h>` have been drilled into the heads of most C programmers, we expect that C++ would only provide different facilities if they seemed a substantial improvement over the interface supported by `<stdio.h>`. Whether or not the C++ I/O library, `<iostream.h>`, is truly new and improved is a judgment best formulated by the reader. We will, however, use the `streams` package in this course, and explain the parts of it we use here.
11.2. NON-OBJECT-ORIENTED FEATURES OF C++

11.2.3.1 The >> and << Input and Output Operators

Contrast the two Hello, World! programs presented below:

/* IN ANSI C */ // IN C++ //
#include <stdio.h> #include <iostream.h>
main() {
    printf("Hello, world!\n");
}
main() {
    cout << "Hello, world!"
         << endl;
}

The most noticeable difference is that output in C++ is handled with an operator, <<, whose arguments aren’t put in parentheses. Also, since << only takes two arguments, but we have several uses of the operator in a row, it must be the case that whatever value is returned by an invocation of << must also be valid as a left-hand argument to <<. The left-hand side of the operator takes “a place to send output”, and the right-hand side takes “the output to send.” The place we specify to send output is called cout, and we may comfortably regard it as equivalent to the stdout of ANSI C. An invocation of << must return whatever its left-hand argument was, then, since we are able to use << several times in a row to send many outputs to the same place:

    cout << "These" << "are" << 6 << "things" << "sent to the screen."
         << endl;

    // Output: These are 6 things sent to the screen.

We now observe another advantage of the << operator — it is overloaded to accept different types of variables we might want to output. We didn’t need to give any special indication that we wanted to show an int, 6, on the screen instead of a string; we just sent it out along with everything else and << took care of printing it. Also, endl isn’t a string, it’s a special argument which tells << to print a newline character (and flush output). By the way, all the data types printable with printf() can be given as arguments to <<, and the usual special characters \n, \t and so on still work in C++ I/O.

C++ also provides a “smart” input operator, >>, which is similar to << with polarity switched:

    int favoriteNum, char favoriteLetter;
    cout << "Please enter your favorite (whole) number and letter:"
         << endl;
    cin >> favoriteNum >> favoriteLetter;

Again we have an operator which uses no parentheses and can be cascaded in sequence. Its left-hand argument is “a place to read input from”, and its right-hand argument is “where to put the input”. C++ provides an input source cin which, as with its C counterpart stdin, defaults to the user’s keyboard or similar input source. Like <<, the input operator >> is overloaded and knows
how to convert the user’s keystrokes into the internal representation of different variable types. In fact, it can even read in strings automatically:

```cpp
char string[80];
cout << "Please enter your name: "; cin >> string;
cout << string << endl; // prints the name
```

Of course, the “place to put” the string has to be a buffer of appropriate size — `>>` will not allocate the memory automatically. Here again, all the types normally handled by `scanf()` can be given as right-hand arguments to `>>`. One of the nicer additions in C++, however, is a special “variable”, `ws`, which automatically consumes whitespace from the input stream:

```cpp
char month[15]; int date, year;
cout << "Type today’s date (month name, then date, then year): ";
cin >> ws >> month >> ws >> date >> ws >> year;
cout << month << ' ' << date << ', ' << year;
```

```cpp
// Prints something like March 12, 1994
```

There is a dizzying assortment of input/output functionality provided through the `>>` and `<<` operators, as well as through `cin` and `cout` (which are, to do a little foreshadowing, actually I/O objects). The discussion here should, however, suffice for the purposes of this course.

### 11.2.3.2 Operator Overloading with `<<` and `>>`

The fact that we can give the C++ input/output operators different kinds of variables without explicitely mentioning what the variable types are usually strikes people as quite natural. After all, we do have some built-in arithmetic operators like `+` and `*`, and we certainly don’t need to be explicit about whether we’re adding `ints` or `floats` — we just put the `+` between two number-like variables and everything works.

Of course, it’s also pretty clear to us that a computer doesn’t add `ints` and `floats` in the same way. Adding two integers is an instruction that is built in to every modern-day computer’s central processor, but floats have a more complex internal representation, so adding them requires the computer to carry out a different algorithm (which may or may not also be built into the hardware). In any event, it’s clear that our commonsense understanding of the similarity between 2 + 2 and 2.5 + 2.5 isn’t really reflected in the work the computer must do to find the answer. So how does the computer decide which version of `+` is appropriate for a given situation?

The answer should be clear to everyone who has read the *Function Overloading* section above. There we said that an operation which is logically similar for different argument types should share the same name across all the functions which implement it — the function’s name should
be overloaded. Plainly, + is an overloaded operator. The compiler chooses which version of the + function to apply to certain arguments based on what types those arguments have. This is true in both C and C++. The difference, however, is that C does not allow a programmer to create her own overloaded functions, or extend the overloading of built-in operators like +. C++ does provide this facility to the programmer, and overloading the << and >> operators for new data types we create can be a powerful way to handle I/O.

Suppose we create a new data type which represents a Harvard ID card. The structure contains some information about the student’s name and year as well as a digitized picture of the student:

```c
typedef struct {
    char lastName[50];
    // Fifty letters suffices for most people
    int graduationYear;
    // Should restrict this to be 199<?
    char pictureBuffer[10240];
    // say the picture uses 10 KB of info
} HarvardID;
```

Now clearly, if we’re really using this structure in a program, we’re going to have to write a function which displays the information stored in a particular variable of type HarvardID, formatting the data nicely and showing the picture in a little window. Suppose we write this function and call it `DisplayID(HarvardID *)`. Calls to this function don’t mesh particularly nicely with the rest of C++ I/O:

```c
HarvardID *myId = BuildID("Jon McAuliffe"); // some init function
cout << "Our example id card looks like this:\n";
    // must call other function now
DisplayID(myID);
    // clumsy break in I/O continuity
cout << endl;
    // finish line and flush
```

What we want to be able to do is give myID as another argument to the C++ << operator, so that the code for I/O looks nicer and, very importantly, nobody using our structure has to remember a special name to print it out. If a programmer wants to display a variable of type HarvardID, she wants to be able to use <<, just like she does for all other output operations. Luckily, C++ allows us to overload <<. The syntax is a little peculiar. A function definition which overloads a built-in operator always has roughly the same syntax: the name of the function is “operator op”, where op is the operator symbol, << in this case.

Thus we may write:
// This gives the same output as our old function DisplayID, but
// uses <<.
//
// ostream &operator<<(ostream &outstream, HarvardId &theID) {
// use << for the components of theID which can be printed
// as text
//
outstream << "ID INFORMATION:\n" << "NAME: "
<< theID.lastName << "\tCLASS: "
<< theID.graduationYear << endl;

// call some function to show the student’s picture in a
// window
//
ShowPicture(theID.pictureBuffer);

// return the stream we used, for cascading
return outstream;
}

Using this overloaded definition of the << operator, we may rewrite our example from above more
simply as

HarvardID *myId = BuildID("Jon McAuliffe"); // some init function

// Now we can just give myID as an argument to << . Look how easy!

cout << "Our example id card looks like this:\n" << myID << endl;

In the above, the arguments to << were of type reference. Consider the tradeoff involved here. Passing an object by value is very inefficient, as it requires copying the whole object. Yet passing objects by reference in the old-fashioned style of C demands careful attention to every address-
of and dereference operator. The compromise is to use the C++ reference type, so that the
arguments appear to be data structures instead of pointers without the efficiency loss of actually
passing by value (they are pointers “under-the-hood”).

We may similarly overload the input operator >> to fill a data type we define with the information needed for its components. In fact, operator overloading is a general improvement over
programming in C. We can overload arithmetic operators to work, say, on a complex number data
type; we can overload the equality test == to test equality of a string data type by comparing each
character in the string, and so on. It is a simple but powerful idea.
11.3 Object Oriented Features of C++

In this section, we will examine the so-called object oriented features of C++.

11.3.1 Encapsulation in the Real World

The object-oriented programming paradigm of C++ is meant to mirror the “real” world. In the world in which we live, we constantly deal with objects: pens, books, grandmothers, telephones, and such. In C++, things like telephone calls, positions in a chess game, and other such non-physical concepts can also be considered as objects.

What do we do with these objects in the real world? Let’s consider a concrete example: a car. When you are driving a car, you can do lots of things including accelerate, decelerate, turn, honk, and so forth. You may also be able to get information back from the car, such as how much gas you have left, and how fast you are going. Of course, you don’t really care how the car works (except when it stops working!). You know that if you press the accelerator the car goes faster, and that if you turn the wheel to the right the car turns to the right. It doesn’t matter whether there’s a gear connected to the steering column connected to the axle or whether the car just magically turns to the right so long as it happens. Similarly, you needn’t worry about the (no doubt fascinating) construction of gas gauges and speedometers, so long as yours continues to function. In more computer-sciency terms, you have an interface to the car (the speedometer, gas gauge, accelerator, and so forth), but you don’t care what the implementation (the rack-and-pinion steering, disc brakes, etc...) is. Why is this separation of interface and implementation a good thing?

Well, for one thing, every time you drive you don’t have to think about all the little things going on. You don’t even have to know what all is going on. It makes life for you, the driver, much simpler. In fact, you might take your car to the shop, have the accelerator replaced with the brand-new super up-to-date environmentally sound accelerator and drive the car home precisely the way you did with the old accelerator. Someone might even sneak into your garage, switch the accelerators back and you might never be the wiser!

So, we say the implementation of the car has been encapsulated. The way it works has been hidden, and can be updated or modified without changing how it is used. Obviously for big software projects, we may decide that we need to modify something, and we certainly don’t want to have to change everything else. In C, we have this idea with functions — if you figure out how to make a function run faster, you can change it to do so, and provided you don’t change its interface, nothing adverse should happen to the rest of the program.
11.3.2 Encapsulation in C++: Interface Specification

In C++, we encapsulate data as well. Suppose the gas gauge shows the amount of gas left in terms of gallons. Suppose further that inside the tank there’s a little device that somehow measures the number of gallons left. So the internal representation of the data (from the car’s point of view) is in gallons. But now, the gauge breaks, and the only one you can get does its measurements in liters. You don’t like liters — you want a gas gauge that shows gallons. Fortunately, there’s a piece of hardware you can add that converts liters to gallons, so the gauge shows gallons, while the car thinks in liters. The data representation has been encapsulated — on the inside it’s different than on the outside, and the inside representation can be changed (reimplemented) without changing the interface.

So, how do we do all this in C++?

Just as in C, we declare a new type called Car. But instead of the keyword typedef that we used in C, we use the keyword class. (Of course typedef is still around in C++, as C++ is a superset of C.) Our first step is to design the class interface.

```cpp
class Car
{
    public: // Everyone in the whole world can use these functions.

        unsigned int WhatsMySpeed(); // Returns speed in miles per hour.
        unsigned int HowMuchGasLeft(); // Returns gallons of gas left.
        void SpeedUp(); // Accelerate.
        void SlowDown(); // Decelerate.
        void TurnRight(unsigned angle); // Turn right angle degrees.
        void TurnLeft(unsigned angle); // Turn left angle degrees.
};
```

This is a legal type declaration in C++. It says that any object of type Car has the functions listed in the declaration. So, if myVW is of type Car, the following are all legal function calls:

```cpp
myVW.SpeedUp();
unsigned int mySpeed = myVW.WhatsMySpeed();
myVW.TurnRight(30);
```

Note that in all these cases we use the name of the object (myVW) followed by a “.” followed by the function call. That’s because every car has its own speed and so forth; it wouldn’t make any sense just to call WhatsMySpeed() without any particular car! But, although we’ve got the interface, we have no implementation! In particular, we haven’t specified the data that the car has
11.3. **OBJECT ORIENTED FEATURES OF C++**

internally. So here’s version 2 of our declaration.

```cpp
class Car {
    public: // Everyone in the whole world can use these functions.
        
        unsigned int WhatsMySpeed(); // Returns speed in miles per
                                        // hour.
        unsigned int HowMuchGasLeft(); // Returns gallons of gas
                                         // left.
        void SpeedUp(); // Accelerate.
        void SlowDown(); // Decelerate.
        void TurnRight(unsigned angle); // Turn right angle degrees.
        void TurnLeft(unsigned angle); // Turn left angle degrees.
    
    private: // This data should be hidden away.
        
        unsigned int gallonsLeft; // Gallons of gas left.
        unsigned int milesPerHour; // Speed of car in mph.
};
```

Now we’ve added the data. Like structs in C, objects in C++ can contain data, but unlike structs they also contain functional interfaces. Don’t worry much about the public and private bit, but the idea is that the data is private to the class (as we said above, the user doesn’t need to know how the car represents the amount of gas left, so long as he or she can look at the gas gauge!), while the functional interface is public so that the driver can look at the gas gauge and speedometer, and use the accelerator and decelerator.

So, in general, a class name ClassName can be declared as follows:

```cpp
class ClassName {
    public:
        
        MemberDeclarations
        
    private:
        
        MemberDeclarations
};
```

Each MemberDeclaration is either a function or variable declaration with the same syntax as in C.
Don’t forget the semicolon at the end of the class!
An object named Identifier of type ClassName can be declared as follows:

```
ClassName Identifier;
```

Member functions of ClassName can be called with the syntax:

```
Identifier.FunctionName(Args)
```

## 11.3.3 Encapsulation in C++: Implementation

We still have to implement the functions, however!
Here’s a first version:

```cpp
unsigned int Car::WhatsMySpeed()
{
    return milesPerHour;
}
```

```cpp
unsigned int Car::HowMuchGasLeft()
{
    return gallonsLeft;
}
```

```cpp
void Car::SpeedUp()
{
    milesPerHour++;
}
```

```cpp
void Car::SlowDown()
{
    milesPerHour--;
}
```
void Car::TurnRight(unsigned angle)
{
  // Talk to the hardware to make some gears turn, or something!
}

void Car::TurnLeft(unsigned angle)
{
  // Talk to the hardware to make some gears turn, or something!
}

So, the functions are defined just like usual C functions, except that the names are preceded by "Car::". In general, if we were defining functions for class X, we would write "X::". So, the general form of a member function of class X is:

```
ReturnType X::FunctionName(Arguments)
{
  // Code goes here.
}
```

### 11.3.3.1 Encapsulation at Work: An Example

Note here how easy it is to change our internal representation.

Suppose we wanted to store the gas remaining in liters, instead of gallons. We just change the name `gallonsLeft` to `litersLeft`, and re-write `Car::HowMuchGasLeft()` as follows:

```
unsigned int Car::HowMuchGasLeft()
{
  return litersLeft * GALLONS_PER_LITER;
}
```

where `GALLONS_PER_LITER` is some constant defined somewhere.

### 11.3.4 Telling Objects Apart

Suppose there are two cars: `MyCar` and `YourCar`. Suppose further that `MyCar`, a slick Porsche, is traveling at 55 mph, while `YourCar`, a beat-up '74 Impala, can only manage 30 mph. Now, the following code is executed:
cout << "I‘m going " << MyCar.WhatsMySpeed() << endl;
cout << "You‘re going " << YourCar.WhatsMySpeed() << endl;
MyCar.SpeedUp();
YourCar.SlowDown();
cout << "Now I‘m going " << MyCar.WhatsMySpeed() << endl;
cout << "Now you‘re going " << YourCar.WhatsMySpeed() << endl;

What happens? You might be afraid that the following could occur:
I‘m going 55
You‘re going 30
Now I‘m going 29
Now you‘re going 29

In other words, you might be afraid that the milesPerHour-- in Car::WhatsMySpeed() would affect some global variable. But, no, the output, will be:
I‘m going 55
You‘re going 30
Now I‘m going 56
Now you‘re going 29

How does this work? Every member function of a class, such as WhatsMySpeed() and SpeedUp() takes a so-called implicit parameter that indicates what object is being referenced. This way the function knows which milesPerHour to change (there is actually a different milesPerHour for every car, just as there would be if we had multiple Car structs in C). This implicit parameter is called the “this pointer”. It‘s as if the functions were declared like:
unsigned int WhatsMySpeed(Car* this);
void SpeedUp(Car* this);
void TurnRight(unsigned int angle, Car* this);

and so forth! Don‘t try to add this parameter yourself — it‘s an IMPLICIT parameter. When you call MyCar.SpeedUp(), the compiler automatically passes along the address of the MyCar object to the Car::SpeedUp() function. The this parameter is in scope inside the member functions, and you may refer to it as necessary. For example, an equivalent, but less clear and therefore inferior, way to implement Car::SpeedUp() would be:
void Car::SpeedUp()
{
    this->milesPerHour++;
}

By the way, we call MyCar and YourCar and any other Cars instances of the class Car. So, the this pointer is a way of letting the class member function know what instance of the class is
11.3. Initialization of Objects: Constructors

11.3.5.1 First Tries

You may have noticed in the example above that I assumed that somehow MyCar magically knew that it was going 55 to start with. There was no initialization. Of course, we do have to initialize objects somehow! You might try something like the following:

```cpp
Car MyCar;
MyCar.milesPerHour = 55;
```

In fact, this would work, if milesPerHour were not private. Since the data is private, we can’t write this sort of thing. We could make the data public, but then we would lose the data encapsulation we strived for so valiantly.

So, you might say, “Why not add a member function?” We could indeed add a function like:

```cpp
void SetSpeed(unsigned int currentSpeedInMPH);
```

to the class declaration, in the public area. This certainly would preserve data encapsulation — we could change the speed variable to keep track of things in kilometers per hour, and then all we would have to do is modify `Car::SetSpeed()` to do a unit conversion.

11.3.5.2 A Better Approach: Constructors

In fact, having a function like this might not be a bad idea, but it still leaves open a problem which has plagued C programmers for decades — what if you forget to do the initialization? Then milesPerHour ends up with garbage in it, but no obvious error may occur for quite some time. C++ introduces special member functions called constructors which are called at the time of object declaration. These constructors always have the same name as the class; that’s how the compiler knows they are constructors. Here’s a new version of Car declaration:
class Car
{
    public: // Everyone in the whole world can use these functions.

    Car(unsigned int InitialSpeed, unsigned int InitialGas);
    // Constructs a Car going InitialSpeed mph with InitialGas
    // gallons of gas left.
    unsigned int WhatsMySpeed(); // Returns speed in miles per
    // hour.
    unsigned int HowMuchGasLeft(); // Returns gallons of gas
    // left.
    void SpeedUp(); // Accelerate.
    void SlowDown(); // Decelerate.
    void TurnRight(unsigned angle); // Turn right angle degrees.
    void TurnLeft(unsigned angle); // Turn left angle degrees.

    private: // This data should be hidden away.

    unsigned int gallonsLeft; // Gallons of gas left.
    unsigned int milesPerHour; // Speed of car in mph.
};

Now, we have all the same function definitions, but add:

Car::Car(unsigned int InitialSpeed, unsignedInt InitialGas)
{
    gallonsLeft = InitialGas;
    milesPerHour = InitialSpeed;
}

Now, to actually create an object of class Car going 55 mph, with 12 gallons of gas left, we write:

Car MyCar(55, 12);

Constructor functions have some unique properties:

1. You can’t call a constructor explicitly. In other words, MyCar.Car(55,12) is illegal at any time.

2. Constructors can’t return a value. Notice that they don’t even return a value of type void (the non-value); they just don’t return anything. For this reason, the declarations begin with
11.3. OBJECT ORIENTED FEATURES OF C++

To review, a constructor is declared and defined just like any other member function, although it cannot have a return type. The constructor is called when the object is initialized (either on the stack as a local variable or on the heap when created with `new`).

### 11.3.5.3 Overloading Constructors

Now, perhaps, most of the cars that you intend to use start at a standstill with a full tank of gas. (Perhaps you are writing yet another oh-so-exciting driving simulation game, and all the racers start at the start line, standing still, but all gassed up...) So, it’s a pain to have all of these

```cpp
Car AnotherCar(0, FULL_TANK);
```

declarations around; you really just want to say

```cpp
Car AnotherCar;
```

and use the full syntax only for exceptional cars that start out moving or with unfull tanks. No problem! We can add another constructor:

```cpp
class Car
{
  public: // Everyone in the whole world can use these functions.

    Car();  // Constructs a Car standing still with a full tank.
    Car(unsigned int InitialSpeed, unsigned int InitialGas);  // Constructs a Car going InitialSpeed mph with InitialGas gallons of gas left.
    unsigned int WhatsMySpeed();  // Returns speed in miles per hour.
    unsigned int HowMuchGasLeft();  // Returns gallons of gas left.

  ...
};
```

In addition to all of the old member functions including the first constructor, we add:

```cpp
Car::Car()
{
  milesPerHour = 0;
  gallonsLeft = FULL_TANK;
}
```

Now, we can just write:
Car AnotherCar();

or

Car AnotherCar;

Of course, we can still say

Car YetAnotherCar(25, 4);

if we want. The overloading of constructors is very handy. In general you may overload any member function. The different versions of the function must have different parameter lists, i.e., they must take different numbers of parameters, or the parameters must be of different types. Note that overloaded functions may not differ solely in their return values.

### 11.3.6 Deinitialization of Objects: Destructors

Another problem which has plagued C programmers since the dawn of recorded time (or at least since Kernighan and Richie) has been forgetting to deallocate resources when no longer needed. The most common case is memory allocated with `malloc()` which is never deallocated with `free()`. You might well have a pointer inside a structure which is a local variable, allocate some memory, and let the structure go out of scope without ever deallocating the memory. Or you might have allocated the structure on the heap, and subsequently deallocated it without remembering to deallocate memory which the structure points to. In C++, this is not a problem!

You may add to a class a function called a destructor, which like a constructor returns no value and must have a special name: `~` followed by the name of the class, so a destructor for `Car` has the name `~Car`. This function will be called whenever an object of class `Car` goes out of scope or is deallocated from the heap, allowing you to do any necessary cleanup.

In our `Car`, example there's nothing obvious for a destructor to do, but you can use your imagination!

### 11.3.7 Friends

We said earlier that function and data members in the `public` section of a class declaration could be accessed by anyone, and that function and data members in the `private` section were accessible only to other members of the class. That's true, up to a point. It's possible that you might want a function outside of the class to have access to the private members; it violates encapsulation, but hey, sometimes you gotta do what you gotta do.

So imagine the following:
11.3. **OBJECT ORIENTED FEATURES OF C++**

```cpp
class MyFriendlyClass
{
    friend void MyVoyeuristicFriend();

    public:

    private:

    void MyVeryPrivateFunctionDontPeek();
};

void MyVoyeuristicFriend()
{
    MyFriendlyClass foo;

    foo.MyVeryPrivateFunctionDontPeek();
}

int main()
{
    MyVoyeuristicFriend();
}

Note that MyVeryPrivateFunctionDontPeek() is private, and that MyVoyeuristicFriend is not a member of MyFriendlyClass, but that none-the-less it is legal for MyVoyeuristicFriend() to call MyVeryPrivateFunctionDontPeek(). That's because MyVoyeuristicFriend() has been declared a friend of MyFriendlyClass. Note that the entire function declaration goes inside the class declaration, but that the definition is external. And since the friend function is not a member (otherwise it wouldn't need to be a friend), you don't put MyFriendClass:: in front of the friend function in it's external definition. And note that the friend function is not invoked with a .; it's not a member function.

11.3.8 Subclasses and Inheritance

Now, you might develop later in your program design uses for various kinds of cars. For instance, you might decide that it would make conceptualization of the situation easier if you had two categories of cars: station wagons, and sportscars. So, in the great tradition of C, you could add an
enum type definition to Car as follows:

```cpp
typedef enum CarKind {station_wagon, sports_car} CarKind;
```

and then add a data member as follows:

```cpp
CarKind carKind;
```

with appropriate member functions for accessing it.

Now, suppose that lots of the member functions for Car behave differently for station wagons and for sports cars. In particular, station wagons speed up and slow down less quickly. And perhaps there are functions like `OpenHatchBack()` that might be appropriate for station wagons, and not for sports cars. You could modify `SpeedUp()` and `SlowDown()` so that they had explicit checks to `carKind`, and that would work OK. But you would have to remember to do that for all the relevant functions, and if you added more types of cars (like luxury sedan, perhaps) you might have to go through and do the same sort of thing all over again. Fortunately, C++ provides a method for dealing for such eventualities.

Look at the following code:

```cpp
class StationWagon : public Car
{
public:

    StationWagon(); // Constructs a station wagon standing still
                     // with a full tank.
    StationWagon(unsigned int InitialSpeed, unsigned int InitialGas); // Constructs a StationWagon going InitialSpeed mph with
                                                                     // InitialGas gallons of gas left.

    void SpeedUp(); // Accelerate.
    void SlowDown(); // Decelerate.
    void OpenHatchBack(); // Open the hatch.
};

StationWagon::StationWagon() : Car()
{

}
In the `StationWagon` class declaration, `: public Car` says that a `StationWagon` is a kind of `Car`. To this end, it has all the same functions and data members as a `Car`. The public members of `Car` are also public in `StationWagon` as well, but the private ones are more than private: they’re invisible!

So now, you can declare a `StationWagon` like this:

```cpp
StationWagon myOldBeatupVW(25, 10);
```

Notice that the constructors for `StationWagon` must call the constructors for `Car` with the `: Car()` syntax. They could do something more in their function bodies, but in this case there’s nothing for them to do.

The redeclarations of existing functions will take precedence, so if we say

```cpp
myOldBeatupVW.SpeedUp();
```

the `StationWagon::SpeedUp()` gets called, not `Car::SpeedUp()`.

For functions that aren’t redeclared the old `Car` versions are used however. And of course, there still is no `OpenHatchBack()` function for ordinary `Cars`.

Now, the particularly clever aspect of all this is that if you have a function, say `Wreck()`, that is declared as:

```cpp
void Wreck(Car* someCar);
```

you can pass `&myOldBeatupVW` to `Wreck()` as follows:

```cpp
Wreck(&myOldBeatupVW);
```
This is because a StationWagon is a Car. Inside Wreck(), all the functions of Car can be used, like:

```cpp
    someCar->SpeedUp();
```

but, since the parameter car was declared as a Car*, not a StationWagon*, the OpenHatchBack() function cannot be called. Unfortunately, as things stand, the function SpeedUp() that is actually used in the above example is the Car::SpeedUp() function, not the StationWagon::SpeedUp() function, despite the fact that the object passed in is actually of type StationWagon.

This problem can be cured by declaring the function SpeedUp() to be virtual in the base class, Car as follows:

```cpp
    class Car
    {
    ... 

        virtual void SpeedUp();
    };
```

Adding virtual has the effect that the correct (derived class) will be called even if the object is accessed through a base class pointer, so the call to SpeedUp() inside of Wreck() will get the function we intended.

Adding the capability for virtual functions is a bigger step than it might seem. It means that the compiler can’t always know what function to call, because it doesn’t know the actual type of the object being passed to the function. We must therefore introduce some runtime system for handling such type resolution dynamically, and let the compiler give the work of resolving calls to virtual functions over to the runtime system. Inevitably, the added work needed for dynamic typing will mean our program runs more slowly than otherwise. So we recognize a much-contested tradeoff between speed and flexibility in object orientation, with some people arguing not to use dynamically-bound functions at all, and others suggesting that every call to an object’s member functions be dynamically bound.

### 11.3.9 Special Member Functions

C++ has some unusual member functions, in addition to the constructor and destructor functions described above. Here we will describe two that are actually operators. Both these operators can be defined only as member functions (actually, member operators), and cannot be defined outside the class to whose objects they apply.
11.3. OBJECT ORIENTED FEATURES OF C++

11.3.9.1 Dererencing: operator->

C++ allows you to create objects that behave somewhat like pointers. Examine the following declaration of class X:

```cpp
class X {
    private:
        Y* my_pointer;
    public:
        ...
        Y* operator->() {
            return my_pointer;
        }
        ...
};
```

Given these definitions, the expression `x->y` is equivalent to `(x.my_pointer)->y`.

In general, if `x` is an object of class `X`, then in the expression `x->y`, the `->` is interpreted as a call to the `X::operator->`, and takes as its value the `Y*` pointer returned. This pointer is then used to reference the `y` that follows the `->` operator. Or in other words, `x->y` is equivalent to `(X::operator->(x))->y)`, more or less.

11.3.9.2 Conversion Operators

Suppose you were writing a class to take the place of the `char*`s that serve as strings in C. Perhaps you wish to provide bounds checking, and built-in functions to do substring search, or some such. So you declare a class `String`. But, you may still need to pass your Strings off to system functions expecting `char*`s; Microsoft Windows, for example, expects strings used as the titles of windows to be passed to it as `char*`s. Suppose `Foo()` is declared as follows:

```cpp
void Foo(char* aString);
```

and suppose further that you have the following declaration:
String myString = "Hello, world.'";

What you want to write is:
   Foo(myString);

But of course you can't: myString is not of type char*.
   You may add to your String declaration as follows:
   class String
   {
   public:

   ...
   
   operator char*();
   
   
   
   
   
   String::operator char*()
   {
   // Return a C-style pointer to the string.
   
   
   
   
   
   This is called a conversion operator. Now, you may use an object of type String anywhere where you would normally use a char*. The operator char*() function will be called at each such use to the conversion.
Chapter 12

Lexical Analysis and Parsing

by Jason Abrevaya, Harry Lewis, and Bob Walton

12.1 An Arithmetic Language

In this section, we will consider a simple arithmetic language. Our first goal is to create a mechanism to “understand” any program in this language. Once we have the ability to “understand” a given program, we will then construct a mechanism which can execute any program in the language. This basic process, although admittedly vague at this point, is the basis for creating an interpreter or compiler for any language — C, Pascal, LISP, etc. We’ll shortly break down the process into some clear-cut steps, but first let’s take a look at the language with which we’ll be working.

Our language will allow a sequence of assignment statements, each of which can contain the four basic arithmetic operations (addition, subtraction, multiplication, division). When a program in this language is executed, we expect that the value of the last variable assigned to will be printed. Consider the following example:

```
a = 21 + 77 * 4;
b = (a + 6) * (a - 3);
temp = (b - a) * -b;
ans = temp + (2 * temp);
```

Each assignment statement in this language is ended with a semicolon; the end-of-program marker is a period. Obviously, nothing more complicated than a simple “variable” can appear on the left-hand side of an assignment statement; we’ll refer to these simple variables as symbolic atoms. Any program, then, is just a series of assignment statements followed by a period. An assign-
ment statement is a symbolic atom followed by an equal sign, some arithmetic expression, and a semicolon.

How do we break an arithmetic expression into simpler building blocks? To do so, we need to first consider the order of arithmetic operations — multiplication and division take priority over addition and subtraction. For instance, when the expression \(21 + 77 \times 4\) is evaluated, the multiplication of 77 and 4 should occur first, and the result should then be added to 21. For this reason, an arithmetic expression should be thought of as a sum or difference of terms. Each term, in turn, should be thought of as a multiplication or division of factors. (The basic idea is that the multiplication and division operations will eventually be performed first in order to determine the values of the terms; the value of an entire expression will be determined by performing last the operations of addition and subtraction on these terms.)

In the last assignment statement of our sample program, the expression is a sum of two terms — \(\text{temp}\) and \((2 \times \text{temp})\). The second of these terms is the multiplication of two factors — 2 and \(\text{temp}\). The first factor, 2, is a number while the second, \(\text{temp}\), is an atom. Factors can also be more complicated than just a number or an atom. Looking at the third assignment statement in our program, we see that the first factor in the arithmetic expression is actually an arithmetic expression itself; the second factor is the negation of an atom.

We can fully describe our language with the following (recursive) grammar:

\[
\begin{align*}
\langle \text{program} \rangle &::= \{\langle \text{asst} \rangle\}^* . \\
\langle \text{asst} \rangle &::= \text{atom} = \langle \text{expr} \rangle \; ; \\
\langle \text{expr} \rangle &::= \langle \text{term} \rangle \{[+,-] \langle \text{term} \rangle\}^* \\
\langle \text{term} \rangle &::= \langle \text{factor} \rangle \{[*] \langle \text{factor} \rangle\}^* \\
\langle \text{factor} \rangle &::= \text{atom} \mid \text{number} \mid (\langle \text{expr} \rangle) \mid -\langle \text{factor} \rangle
\end{align*}
\]

In this grammar, \{\text{blah}\}* means “zero or more occurrences of \text{blah}.” \(a|b\) and \([a, b]\) both mean “either \(a\) or \(b\) (but not both).” An \text{atom} is considered to be any sequence of alphabetic characters (\(a, \text{temp}, \text{ans}, \text{etc.}\)), and a \text{number} any sequence of digits (0, 4, 77, etc.). (To be completely formal about the grammar, we should really write out the rules which dictate what constitutes an atom and a number.)

We need to distinguish between two types of entities in our grammar — terminals and non-terminals. The non-terminals are those elements which can be further expanded by some grammar rule; that is, each element appearing on the left-hand side of a rule is a non-terminal. In this grammar, then, there are five non-terminals (\langle \text{program} \rangle, \langle \text{asst} \rangle, \langle \text{expr} \rangle, \langle \text{term} \rangle, \langle \text{factor} \rangle). Each of the other elements of the grammar are terminals — symbolic atoms, numbers, arithmetic operators, left and right parentheses, equal sign, semicolon and period.

(Note: The division operator, unlike the three other arithmetic operators, is not associative. If there are multiple divisions in the same term, we’ll use the convention that the division operators
are used left-to-right. For instance, the term \(7/3/2\) has a value of 1 (the 7 is divided by the 3 first, giving a result of 2; this 2 is then divided by 2, giving a result of 1).)

12.2 The Building Blocks of Language Analysis

Above, we indicated that two main steps — understanding and execution — are necessary to interpret a language. However, an interpreter is usually broken down into three levels. The “understanding step” takes place on two levels — the lexical level and the parse level. The “execution step” occurs on the semantic level.

At the lexical level, the input stream (i.e., the program) is broken up into units, called lexemes or tokens, that will be significant to the parser. Lexical analysis picks apart those lexemes which are distinguishable in the language’s grammar. (In our example, the lexemes are simply the terminals of the grammar.)

At the parse level, the stream of lexical tokens is analyzed according to the rules of the grammar. Strictly speaking, a parser returns a true or false answer to the question: “Is the input valid according to the grammar of the language?”

At the semantic level, actions are taken in accordance with the intended meaning of the grammatical constructs. For example, if \(<expr>\) is matched to \(<term> + <term>\) by the rules of our grammar, it means that the two \(<term>\)s have numeric values that should be added to yield a numeric value for the \(<expr>\). For our language, then, the actions in the semantic level involve arithmetic calculations and, eventually, output of the final assignment statement’s value. The semantics of language analysis, however, can take many forms. In a compiler, for instance, we might generate assembly-language code at the semantic level rather than performing explicit calculations.

To understand how the three levels of language analysis interconnect, consider the following diagram:
12.3 The Lexical Analyzer

Our lexical analyzer (or scanner) should be able to look at a stream of input and break this input up into tokens. For our language, these tokens are simply the terminals of the grammar. We can number them as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>symbolic atom</td>
</tr>
<tr>
<td>1</td>
<td>number</td>
</tr>
<tr>
<td>2</td>
<td>arithmetic operator</td>
</tr>
<tr>
<td>3</td>
<td>assignment token</td>
</tr>
<tr>
<td>4</td>
<td>EOF token</td>
</tr>
<tr>
<td>5</td>
<td>LParen token</td>
</tr>
<tr>
<td>6</td>
<td>RParen token</td>
</tr>
<tr>
<td>7</td>
<td>semicolon token</td>
</tr>
</tbody>
</table>

12.3.1 The Grammar

The following grammar is for our language.

```
Program ::= Declaration
  | Declaration Assignment
Declaration ::= Type ID := Expression
Assignment ::= ID := Expression
Expression ::= Term | Term + Term | Term - Term | Term * Term | Term / Term |
Term ::= Factor | Term * Factor | Term / Factor
Factor ::= Number | ( Expression ) |
Number ::= Digit | Number Digit |
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```
12.3. THE LEXICAL ANALYZER

Unfortunately, it’s not sufficient to return just the tag associated with a token. Only the last five tokens in the list above are fully determined by their numerical tag; that is, we know that a LParen token is a left parenthesis, an EOF token is a period, etc. More information is needed for the first three types of tokens. An atom token should have a name associated with it, a number token should have a value associated with it, and an operator token should have the appropriate operation associated with it.

Symbolic atoms are the most complex of the tokens in our list since we need to initialize and alter the values of atoms according to the assignment statements of a program. In addition to a print name, then, an atom token also needs to be associated with an initialization flag (to tell whether any assignments have been made to the atom) and a value. We’d also like to keep at most one copy of each atom token; that is, if an atom is seen for the second or third time in the program, we want to refer to the token which was already set up for that atom. To do this, we can string together the atom tokens in a linked list so that a repeated symbolic atom can be matched against one previously read.

Rather than having a different type of data structure for each lexeme, our approach will be to use a single data structure, even if not all the fields in the structure are needed for all types of lexemes. Here is the C code for setting up our lexeme data structure:

```c
/* types of operators: */
typedef enum { OP_NEG, OP_PLUS, OP_MINUS, OP_MULT, OP_DIV } OP_TYPE;

/* lexeme types: */
typedef enum {
    T_SYM, /* Symbolic atom token */
    T_NUM, /* Number token */
    T_OP, /* Operator token (+, -, *, /) */
    T_ASST, /* = */
    T_EOF, /* End-of-file (.) */
    T_LPAREN, /* ( */
    T_RPAREN, /* ) */
    T_SEMICOLON /* ; */
} LEXTYPE;
```
typedef struct lexeme {
    LEXTYPE type; /* one of the T_... */
    char *pname; /* for symbolic atoms: */
    struct lexeme *nextatom; /* for symbolic atoms: */
    int value; /* for numbers, symbolic*/
    int inited; /* for symbolic atoms: */
} lexeme, *lexptr;

With these definitions, consider what some of our token representations might look like:

<table>
<thead>
<tr>
<th>type</th>
<th>T_LPAREN</th>
<th>T_OP</th>
<th>T_NUM</th>
<th>T_EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>pname</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nextatom</td>
<td></td>
<td>OP_TIMES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>inited</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

atomlist

The representations in the first row of pictured tokens are very straightforward. A left parenthesis and an EOF marker can be represented with just their LEXTYPE tag. An operator needs to have its operation specified in the value field, while a number needs to have its integer value specified in the value field.

The representation of symbolic atoms is a little more involved. Notice first the pointer atomlist, which is of type lexptr. This pointer denotes the beginning of our linked list of symbolic atoms; at the outset, we want to initialize atomlist to NULL to indicate that no symbolic atoms have been added to the linked list as of yet. In the pictured list, there are two symbolic atoms, named a and temp. Looking at the token structure for the atom a, we see that it has been initialized and that
it currently has a value of 17; the nextatom field of the structure points off to the next atom in the list, temp. The structure of temp’s token indicates that it has yet to be initialized; moreover, the NULL value in the nextatom field indicates that temp is the last atom in the linked list.

We’ll need to have atomlist as a global variable in our program. In addition, it will also be helpful to store the character stream input into a global buffer. Here is a file containing the necessary global-variables for our lexical analyzer:

```c
/* Global Variables: */
** File: globals.c
** Project: Arithmetic Language Interpreter */

#include "data.h"
#include "extern.h"

char inputbuf[BUF_SIZE]; /* buffer to store the input */
lexptr atomlist; /* pointer to list of symbolic */

BUF_SIZE is a constant equal to the maximum number of characters allowed in any <program> (plus 1 for a ‘\0’ character). We need to define that constant, and we also need to put the data declarations for OP_TYPE, LEXTYPE, lexeme, and lexptr given above somewhere. We create a file for this purpose:

/* Constants and Data Structures */
** File: data.h
** Project: Arithmetic Language Interpreter */

#define BUF_SIZE 1000 /* size of input buffer */
#define SYMBOL_SIZE 100 /* size of largest symbol */
typedef enum { FALSE, TRUE} boolean;
```
/* character constants:
 */
#define PLUS_SIGN '+'
#define MINUS_SIGN '-'
#define TIMES_SIGN '*'
#define DIVIDES_SIGN '/'
#define EQUAL_SIGN '='
#define LEFT_PAREN '('
#define RIGHT_PAREN ')' 
#define SEMICOLON ';
#define EOF_MARKER '.'

(declarations for OP_TYPE, LEXTYPE, lexeme, and lexptr go here)

With our data structures and global variables set up, we can start writing the lexical analyzer, which will look through the supplied input, character by character, in an attempt to convert the input into tokens. At any point in the scanning process, we can keep track of our current location in the input through the use of a char pointer. All but two of our tokens (T_SYM and T_NUM being the exceptions) can be recognized when a single character is encountered; for instance, if our scanner sees a semicolon, it should know to return a token of type T_ASST.

Scanning numbers and symbolic atoms is a bit more difficult. Any digit in the input signals the beginning of a number, but since a number can be any sequence of digits, we have to continue scanning past this initial digit. We keep scanning, character by character, until reaching a character which is not a digit. Any such character (even a space or carriage return) signals the end of the numeric token. An additional complication with numbers is that we need to be able to compute the number's value (to be placed in the value field) while scanning through the digits. The scanning process for symbolic atoms is similar to the one for numbers. An alphabetic character signals the beginning of an atom token, and scanning should continue until reaching a non-alphabetic character. Rather than computing a value (as was done in the case of a numerical token), we want to keep track of the alphabetic characters read for the atom so that we can attach the proper string to the pname field of the token structure.

Here is the complete C code for our scanner:
/* Lexical-analysis routines
**
** File: lex.c
** Project: Arithmetic Language Interpreter
*/

#include "data.h"
#include "extern.h"

static char *sp; /* current location in inputbuf */
static lexptr curlex; /* one-lexeme buffer */
static char lexflag; /* does curlex contain next */
    /* lexeme? */

static lexptr readlex(void);

static lexptr initlex(void)
{
    lexflag = FALSE;
    sp = inputbuf; /* initialize input buffer pointer */
    atomlist = NULL; /* initialize linked list of */
    /* symbolic atoms */
}

void ungetlex(void)
{
    lexflag = TRUE;
}
/**
 * getlex -- get next lexeme from token stream
 */
lexptr
getlex(void)
{
    if (!lexflag)
        curlex = readlex();
    lexflag = FALSE;
    return curlex;
}

/*
 * readlex -- read a new lexeme from input
 */
static lexptr
readlex(void)
{
    lexptr l;
    char c, *p2, tbuf[Ssymbol_SIZE];
    int n;

    l = (lexptr) safe_malloc(sizeof (lexeme));
    /* storage for new token */
    if (l == NULL) /* out of storage? */
        error("out of memory");

    while (isspace(c = *sp++)); /* ignore white space */

    switch (c)
    {
    case EOF_MARKER:
        l->type = T_EOF;
        return l;
    case PLUS_SIGN:
        l->type = T_OP;
        l->value = OP_PLUS;
        return l;
    }
case MINUS_SIGN:
    l->type = T_OP;
    l->value = OP_MINUS;
    return l;

case TIMES_SIGN:
    l->type = T_OP;
    l->value = OP_MULT;
    return l;

case DIVIDES_SIGN:
    l->type = T_OP;
    l->value = OP_DIV;
    return l;

case LEFT_PAREN:
    l->type = T_LPAREN;
    return l;

case RIGHT_PAREN:
    l->type = T_RPAREN;
    return l;

case SEMICOLON:
    l->type = T_SEMICOLON;
    return l;

case EQUAL_SIGN:
    l->type = T_ASST;
    return l;

}

if (isdigit(c))
{
    sp--;
    n = 0;       /* gobble up decimal number */
    while (isdigit(c = *sp++))
        n = n * 10 + (c - '0');
    sp--;       /* correct overshoot -- */
                 /* back up character pointer */
As our scanner is written above, the function `getlex` can be called to retrieve the next token from the input stream. The scanner is, however, still a bit incomplete. Notice that there is a call to a function named `lookup`. This function looks through the linked list of symbolic atoms to find an existing copy of the atom; if no such copy is found, a new atom is created. This function is part of the semantic routines since it is really within the realm of execution rather than the realm of scanning.

To allocate memory for a token, we use a routine `safe_malloc` which checks that there is memory available for the allocation. There is also a call to an outside `error` function within our scanner. This function reports an appropriate error message and exits the program immediately. These utility functions are put in file by themselves:
/* Utility functions
**
** File: utilities.c
** Project: Arithmetic Language Interpreter
*/

#include "data.h"
#include "extern.h"

#define EXIT_FAILURE 1
	/* Return error code = 1 from program failure */

/*
* safe_malloc -- version of malloc that checks that memory is available for allocation
*/
void *
safe_malloc(int size)
{
    void *result = malloc(size);

    if (result == NULL)
        error("out of memory");
    return result;
}

/*
* error -- prints appropriate error message and exits program
*/
void
error (char *msg)
{
    printf("error: %s\.n\n", msg);
    exit(EXIT_FAILURE);
}
12.4 The Parser

With a lexical analyzer at our fingertips, we have the ability to transform a stream of input into a stream of tokens. Our next goal is to make some sense out of this token stream. Specifically, we want to use our grammar in order to see if and how these tokens follow the syntactic rules of the language. This process is known as parsing.

We want to somehow match up the entire input (converted into a token stream) with the grammar rule for a `<program>`. But how do we even start? A good first step is to just look at the first token in a program’s token stream. If this token is of type T_EOF, we have a trivial program with no assignment statements. If the token is of any other type, we know there must be at least one `<asst>`. After all, the grammar rule for `<program>` says that there are zero or more assignment statements, followed by a period; since we didn’t see the period as the first token, it can’t be the case that there are zero assignment statements.

Looking at the rule for `<asst>`, our token must be a symbolic atom; if it’s not, the grammar has been violated — a syntax error has occurred. In addition, the next token in the stream must be of type T_ASST (i.e., an equal sign). Then, the rule tells us, we should try to continue matching tokens, as they appear in the stream, to the rule for `<expr>`.

We can continue this process, reading in one token at a time and checking which alternative from a grammar rule can possibly be matched. Let’s consider a specific example of a program to see how it gets parsed:

```
a = 32 + 21;
```

To clarify things, we can represent the parsing process by a parse tree. At the root (top) of the tree, we have the non-terminal `<program>`. The branches that extend from any element of the tree indicate how its grammar rule is satisfied:
12.4. THE PARSER

The branches emanating from \texttt{<program>} indicate that the program consists of a single assignment statement followed by a period. The branches from \texttt{<asst>} indicate that a symbolic atom is being assigned to a single expression. This expression is simply the sum of two terms. Each of these terms is a factor and, in particular, they are both numbers.

If we examine our grammar a bit more closely, we can see a very special property which it exhibits. At any point in the parsing process, we can tell exactly which alternative of a grammar rule should be chosen by looking at the next token in the stream. That is, for any given input, there is only one possible way in which the token stream can match the grammar; and, if it matches, the matching is always evident from the next token in the stream.

Consider the rule for \texttt{<factor>}. By looking at the next token, we know exactly which of the four alternatives on the right-hand side of the rule should be chosen. If the token is of type \texttt{T_SYM}, the appropriate alternative is “atom”; for type \texttt{T_NUM}, we’d choose the “number” alternative; for type \texttt{T_LPAREN}, we’d choose the \texttt{(<expr>)} alternative; and, finally, for type \texttt{T_OP} (if the operator is a minus sign), we’d choose the alternative \texttt{<factor>}. If none of these four cases hold, we know that a syntax error must exist within the input.

As another example, consider what happens when we parse an \texttt{<expr>}. After a series of tokens has been matched against a \texttt{<term>}, the next token tells us precisely the shape that the expression will take. If the token is not an addition or subtraction operator, we know that the \texttt{<expr>} has been fully matched already. If, however, it is one of these two operators, we should try to match tokens from the stream (after the operator) against another \texttt{<term>}. After this \texttt{<term>} is parsed,
we again look for an addition or subtraction operator as the next token and proceed in the same
manner.

To convince yourself of the special property of our grammar, it might be a good exercise to
actually go through the aforementioned two-line program and follow the branches of the parse tree.
You’ll notice that each alternative (branch) from a given rule of the grammar is fully determined
by the stream of tokens. There are never two possibilities for satisfying a rule if the next token
is known. Furthermore, there is never a need to know anything about the tokens beyond the next
token.

We seem to be making quite a fuss over this property — but for good reason. This property
(the ability to parse using one-token lookahead) allows for a very straightforward implementation
of a parser. The type of parser which can be built for a grammar with this property is known as a
recursive-descent parser. (Such a parser can be built for certain other grammars as well, but we’ll
be confining ourselves to only those grammars which are determined by one-token lookahead.)

The basic idea behind writing a recursive-descent parser is that there will be a function associ-
ated with each non-terminal in the grammar (or, in other words, with each rule of the grammar).
Remembering that a non-terminal appears on the left-hand side of a rule, the function associated
with a non-terminal becomes almost a direct translation of the right-hand side of the appropriate
rule. Consider, for example, how we would parse the rule for an \texttt{<expr>}:

\[
\texttt{<expr> ::= <term> \{[+,-] <term>\}^*}
\]

Here is a simple sketch (in C pseudo-code) of what the corresponding function in the parser would
look like:

```c
expr()
{
    term();
    while (next token is + or -)
    {
        term();
    }
}
```

The actual code for this function will be a bit more complicated since we’ll want to keep track of
numeric values throughout the program in order to come up with a final return value at the end.
(The routines that perform the necessary arithmetic operations will be part of the semantics, but
we’ll need to make calls to these routines from our parser.) In addition, there is some bookkeeping
which needs to be done. When we check to see if the next token is an addition or subtraction
operator, we need to put the token back into the stream in the case that it’s not one of these two
operators (so that the token isn’t just skipped).

The fact that our parser is “recursive” follows from the recursive nature of our grammar. For
instance, the function corresponding to \texttt{<factor>} will clearly need a recursive call on itself (for
12.4. THE PARSER

the alternative \(-\text{<factor>}\) and a recursive call on the function for \(<\text{expr}>\) (for the alternative \(<\text{expr}>\)).

Here is the complete C code for our recursive-descent parser:

```c
/* Parse routines */
/** * File: parse.c
** * Project: Arithmetic Language Interpreter */

/* The grammar for the arithmetic language is as follows: */
/** * <expr> -> <term> \{ [+,-] <term> \}*
** * <term> -> <factor> \{ [*,/] <factor> \}*
** * <factor> -> <atom> | <number> | ( <expr> ) | -<factor>
** * <asst> -> <atom> = <expr>;
** * <program> -> { <asst> }*

** The parser calls external semantic routines, which can be changed without changing the parser. */

#include "data.h"
#include "extern.h"

static int expr (void);
static int term (void);
static int factor (void);
static int asst (void);
```
/**
 * expr -- parses an arithmetic expression
 */

static int
expr(void)
{
    int v, v2;
    char c;
    lexptr l;

    v = term();
    while (((l = getlex())->type) == T_OP)
        &&
        (l->value == OP_PLUS || l->value == OP_MINUS)
    {
        v2 = term();
        if (l->value == OP_PLUS)
            v = sum_sem(v, v2);
        else
            v = diff_sem(v, v2);
    }

    /* an extra token was retrieved to see if it was
    ** + or -
    */
    ungetlex();
    return v;
}
/ * term -- parses a term
 */

static int term(void)
{
    int v, v2;
    char c;
    lexptr l;

    v = factor();
    while ( (((l = getlex())->type) == T_OP) 
           && 
           (l->value == OP_MULT || l->value == OP_DIV) )
    {
        v2 = factor();
        if (l->value == OP_MULT)
            v = prod_sem(v, v2);
        else
            v = div_sem(v, v2);
    }

    /* an extra token was retrieved to see if it was
    ** "*" or "/"
    */
    ungetlex();
    return v;
}
/ * factor -- parses a factor (includes case for symbols and numbers) */

static int factor(void)
{
    leexpr l;
    int v;

    l = getlex();
    switch (l->type)
    {
        case T_SYM:
            if (l->inited)
                return sym_sem(l);
            else
                error("use of uninitialized " "variable");
        case T_NUM:
            return num_sem(l);
        case T_LPAREN:
            v = expr();
            if (getlex()->type == T_RPAREN)
                return v;
            error("unmatched parentheses");
        case T_OP:
            if (l->value == OP_MINUS)
            {
                v = factor();
                return neg_sem(v);
            }
            break;
    }
    error("unrecognizable factor");
}
12.4. THE PARSER

```c
/*
 * asst -- parses an assignment statement
 */
static int asst(void)
{
  lexptr l, l2;
  int v;

  if ((l = getlex())->type == T_SYM
      && (l2 = getlex())->type == T_ASST)
  {
    v = asst_sem(l, expr());
    if ((l2 = getlex())->type == T_SEMICOLON)
      return v;
    error("invalid right hand side of assignment "
           "statement");
  }
  error("invalid assignment statement");
}

/*
 * program -- parses a program and returns the value of its
 * last assignment statement
 */
int program(void)
{
  int v = 0; /* return zero if no assignment */
  /* statements */
  init_sem();
```
while (getlex()->type != T_EOF) {
    ungetlex();
    v = asst();
}
return v; /* return the value of the last assignment statement */

Our parser calls upon external semantic routines (like sym_sem, asst_sem, prod_sem, etc.) which can be changed without changing the parser. Whenever a syntax error is encountered, error is called with an appropriate error message.

12.5 Semantics

We’ve now completed the first two boxes from our language-analysis diagram, leaving only semantics unfinished. Our semantic routines will directly determine exactly what our program will do. After all, the scanner and parser only serve to analyze the input and to see if and how it fits the grammar.

We’re actually only trying to write an interpreter for our arithmetic language. That is, we want to interpret a given program and execute its statements. The semantic routines to take care of the arithmetic operations, then, are quite simple — the product of two factors is arrived at by multiplying the values of the factors, the sum of two terms is arrived at by adding the values of the terms, etc. In order to take care of assignment statements, we need a semantic routine which will change the value of the appropriate symbolic atom.

Here is the remarkably simple code for our semantics:
/* Semantic routines */
** File: semantics.c
** Project: Arithmetic Language Interpreter */

#include "data.h"
#include "extern.h"
12.5. SEMANTICS

/*
 * init_sem -- no initialization
 */
void
init_sem(void) {} 

/
 * sum_sem -- addition semantics
 */
int
sum_sem(int v1, int v2)
{
    return v1 + v2;
}

/
 * diff_sem -- difference semantics
 */
int
diff_sem(int v1, int v2)
{
    return v1 - v2;
}

/
 * prod_sem -- product semantics
 */
int
prod_sem(int v1, int v2)
{
    return v1 * v2;
}
/ * div_sem -- division semantics  
  */  
  int  
  div_sem(int v1, int v2)  
  {  
    return v1 / v2;  
  }  

/*  
* sym_sem -- semantics of symbolic atom  
*/  
  int  
  sym_sem(lexptr l)  
  {  
    return l->value;  
  }

/*  
* num_sem -- semantics of numeric atom  
*/  
  int  
  num_sem(lexptr l)  
  {  
    return l->value;  
  }

/*  
* neg_sem -- semantics of unary negation  
*/  
  int  
  neg_sem(int v)  
  {  
    return -v;  
  }
/* asst_sem -- semantics of an assignment statement */
int asst_sem(lexptr l, int v)
{
    l->value = v;
    l->inited = TRUE;
    return v;
}

/*
 * sym_init -- initializes a new symbolic atom
 * (invoked from lookup)
 */
static void
syminit (lexptr l, char *p)
{
    l->type = T_SYM;
    l->pname = (char *) safe_malloc
        (sizeof(char) * (strlen(p) + 1));
    if (l->pname == NULL)
        error("out of memory");
    strcpy(l->pname, p);
    l->inited = FALSE; /* no value for atom yet */
    l->nextatom = atomlist; /* insert atom into list */
    atomlist = l;
}
CHAPTER 12. LEXICAL ANALYSIS AND PARSING

/*
 * lookup -- looks for symbolic atom in symbol table
 * (a linked list). If it's not there, adds it and
 * returns address of new atom; if it is there,
 * returns address of atom in the list.
 */
lexptr
lookup(char *p, lexptr newatom)
{
    lexptr theLex;

    for ( theLex = atomlist;
          theLex != NULL;
          theLex = theLex->nextatom)
    {
        if (!strcmp(p, theLex->pname))
            { /* If it's in the list, release the
               ** space allocated, and return address
               ** of the existing atom.
               */
                free(newatom);
                return theLex;
            }
    }

    /* if it's not in the list, initialize the new atom
    */
    syminit(newatom, p);
    return newatom;
}

Recall from our scanner the call to lookup. This semantic routine is used to create a new symbolic
atom if it doesn’t already exist in the linked list of symbolic atoms. To “create” a symbolic atom,
we retrieve enough memory to fit its print name, turn its initied flag off, and insert it into the
linked list.

A logical question is why we even have the function init_sem. A call to this routine oc-
curs from the parser, but the function clearly does nothing. The fact is that we could do without
init_sem since our semantic routines don’t require any form of initialization. However, if we
were interested in writing some other type of semantic routines that did require an initialization
routine, we could do so without having to alter our parser since it already allows the possibility of initialization. (For instance, if we wanted to write a compiler instead of an interpreter, our semantics would change quite a bit. We would generate the appropriate source code within each semantic routine rather than computing a numeric value; in addition, it turns out that a couple of variables would have to be initialized at the outset.)

Our interpreter is just a single step away from being completed. We need a driver routine (main) which can take a stream of input and set the language-analysis process in motion:

```c
#include "data.h"
#include "extern.h"

static int read_program(void);

main()
{
    int value;       /* value of interpreted program */

    while (read_program())
    {
        initlex();
        value = program();
        printf("\nValue: %d\n\n", value);
    }
}
```

/* Main routines
 **
 ** File:       interpreter.c
 ** Project:   Arithmetic Language Interpreter
 */
read_program -- reads a program as input (the end of a program is signalled by a period)
*/

static int
read_program(void)
{
    char *p; /* pointer to input buffer */
    boolean eof = FALSE; /* end of file marker has been read */

    printf("Please enter your program, ", "ending with a \'%c'.\n", EOF_MARKER);
    printf("Type \'%c' alone to quit:\n", EOF_MARKER);

    p = inputbuf;
    do /* append the input lines into the input buffer */
    { /* put pointer at end of the input read so far */
        fgets(p, BUF_SIZE - (p - inputbuf), stdin);
        /* second arg is (# of chars left in buffer) */

        /* stop when period is entered or buffer is full */
        /* Anything after a period will be ignored */

        if (!eof)
            error("input buffer overflow");
        /* does not return */
        } while ((p < inputbuf + BUF_SIZE - 1) && !eof);
    for ( ; (*p != '\0') && !eof; p++)
        if (*p == EOF_MARKER)
           _eof = TRUE;

    /* stop when period is entered or buffer is full */
    /* Anything after a period will be ignored */

    if (!eof)
        error("input buffer overflow");
    /* does not return */
12.5. SEMANTICS

/* skip over white space */
for (p = inputbuf; isspace(*p); p++);

return (*p != EOF_MARKER);
}

Lastly, we need a .h file to contain all the external declarations referenced in various files, and then we are done:

/* External variables and function prototypes */
**
** File: extern.h
** Project: Arithmetic Language Interpreter
*/

/* Import packages:
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* Function Prototypes: */
void error (char*); /* interpreter.c */
void *safe_malloc (int);
void initlex (void); /* lex.c */
void ungetlex (void);
lexptr getlex (void);
int program (void); /* parse.c */
void init_sem (void); /* semantics.c */
int sum_sem (int, int);
int diff_sem (int, int);
int prod_sem (int, int);
int div_sem (int, int);
int sym_sem (lexptr);
int num_sem (lexptr);
int neg_sem (int);
int asst_sem (lexptr, int);
lexptr lookup (char *, lexptr);

/* Globals: */
*
extern char inputbuf[];
extern lexptr atomlist;

12.6 Parsing Projects

12.6.1 Regular Expressions and Pattern-Matching

12.6.1.1 Strings and Regular Expressions

Most programming languages support some concept of a string, which is a sequence of zero or more characters. For example, "a", "hot", "dog", and "Computer Science" are all strings made up of upper and lower case characters. Every string has a length, which is the number of characters in the string. (The string consisting of no characters, called the empty string, has length zero and is denoted "\".\") The concatenation of two strings is the string consisting of the characters of the first string in their original order followed by the characters of the second string in their original order. For instance, the concatenation of "hot" and "dog" is "hotdog".

Following the C language conventions, we will delimit characters by single quotes and strings of characters by double quotes. Keep in mind the difference between 'a' and "a". The first is just a character, while the second is a string of length 1 whose first (and only) character is 'a'.

A regular expression is a notation used to describe a particular class of strings. For our purposes, a regular expression will be made up from the alphabetic characters a, b, . . . , z, the digits 0, 1, . . . , 9, and certain syntactic metacharacters: #, ?, |, *, +, (, ).

A single alphanumeric character is a regular expression, as are # and ?. Larger regular expressions can be built up recursively as follows: if \(\alpha\) and \(\beta\) are regular expressions, then \((\alpha \mid \beta)\),
(αβ), α*, and α+ are also regular expressions. These four constructions are called, respectively, the *union* of α and β, the *concatenation* of α and β, the *closure* of α, and the *positive closure* of α.

A string is in the class of strings described by a regular expression if the regular expression *matches* that string. The six rules for matching are very simple:

1. Any alphanumeric character matches itself (e.g., the regular expression a would match the string "a").

2. ? matches any single character. For example, ? matches "a".

3. # matches the empty string, "".

4. (α | β) matches any string that matches either α or β (e.g., (a | b) would match both "a" and "b", but not "c").

5. (αβ) matches the concatenation of any string that matches α with any string that matches β (e.g., (ab) matches only the string "ab").

6. α* matches the string formed by concatenating any number (including zero) of strings that α matches. For example, a* matches any string consisting of only character 'a' — i.e., "", "a", "aa", etc. (Note that a* matches "" because the concatenation of zero strings is the empty string.) The expression (a | b)* matches strings containing only the characters 'a' and 'b', while a (a | b)* b denotes any such string that begins with 'a' and ends with 'b'.

7. α+ matches the string formed by concatenating a non-zero number of strings that α matches. For example, a+ matches any non-empty string which consists of only the character 'a' — i.e., "a", "aa", etc. (Note that a+ is equivalent to (aa*)).

We can somewhat relax the rules that describe regular expressions since some of the parentheses used in building up a regular expression can be redundant. (For instance, ((ab) (cd)) could be written more simply as abcd.) Here is a grammar for regular expressions (REs, for short) which eliminates some redundant parentheses:

\[
\begin{align*}
<RE> &::= <RT> \{ | <RT> \}^* \\
<RT> &::= <RF> \{ <RF> \}^* \\
<RF> &::= ( <RE> ) \mid <RF>^* \mid <RF>^+ \mid <alfanum> \mid \# \mid ? \\
<alfanum> &::= a \mid b \mid \ldots \mid z \mid 0 \mid 1 \mid \ldots \mid 9
\end{align*}
\]

Notice the difference between symbols of the regular-expression language and characters which are used to explain the grammar. The *'s in the first two rules are part of the metalanguage we are
using to write grammars; they mean “any number of,” so that an \(<RE>\) could be an \(<RT>\), or \(<RT>\mid<RT>\), or \(<RT>\mid<RT>\mid<RT>\), etc. The * in the third rule is a lexical token that can be one of the symbols in a regular expression. Likewise, the | in the third and fourth rules is a metacharacter, while the \(\mid\) in the first rule is a lexical token.

Consider the following regular expressions (and the classes of strings against which they match):

\[(a\mid ba)^+\]

is an \(<RF>\) (and also an \(<RT>\) and \(<RE>\)). This denotes non-zero length strings of a’s and b’s in which each ‘b’ is followed by an ‘a’ (such as “abaaba”).

\[(a\mid ba)\ast (\# \mid b)\]

is an \(<RT>\) (and also an \(<RE>\)). This denotes strings of a’s and b’s in which there are no consecutive b’s (such as the example above, or "aababab").

\[(a\mid ba)* (\# \mid b) \mid (b \mid ab) \ast (a \mid \#)\]

is an \(<RE>\). This denotes strings of a’s and b’s in which there are not both consecutive a’s and consecutive b’s (such as the two examples above, or "a", "b", and ").

\[(?a)\ast\]

is an \(<RF>\) (and also an \(<RT>\) and \(<RE>\)). This denotes strings of non-zero even length in which every second character is an ‘a’.

There is a rich mathematical theory related to the regular expressions and classes of strings they denote, and there are many practical applications. Regular expressions, and minor variations thereof, are commonly used in computer command languages to denote “all the strings [file names, etc.] of the following general form.” One utility for regular expression pattern-matching, called \texttt{egrep} (for “Extended General Regular Expression Parser”), is available under the UNIX operating system. This project involves writing a version of \texttt{egrep}’s core — in other words, to solve the problem of determining whether a particular string is in the class of strings denoted by a given regular expression. Here is an example of the kind of behavior your program should exhibit (user input is in a special type font):

\textbf{Please enter a regular expression:}\n\[(a\mid ba)\ast (\# \mid b) \mid (b \mid ab) \ast (a \mid \#)\]

\textbf{Please enter a string:}\n\texttt{aababaab}

That string matches the expression.
\textbf{Do you want another string for the same expression? y}
12.6. PARSING PROJECTS

12.6.1.2 Non-Deterministic Finite Automata

A non-deterministic finite automaton, or NDFA, is a formal representation of an imaginary machine which reads a string of characters and at the end of the string either accepts or rejects the string. In some sense, then, an NDFA also describes a class of strings because it divides the set of all strings into two component sets: the strings it accepts, and the strings it rejects.

An NDFA can be represented by a directed graph. The nodes represent states of the imaginary machine. The edges represent possible transitions from one state to another. All edges are labeled by either a character or by "", the empty string. Two states of the machine are special: the initial state and the final state. (The initial state and the final state may be the same.) The machine begins in the initial state with its “reading head” at the first character of its input string. It operates by following edges from state to state. A machine can legally follow an edge if the edge is labelled by the empty string or if the character under the head is the same as the character that labels the edge. After following an edge labelled by a character, the “reading head” moves to the next character in the input string. It’s possible that there may be more than one legal transition from a given state; any of these transitions can be selected and followed. (This is the “non-deterministic” part of the machine. A machine that never has two possible moves to choose between is said to be

Please enter a string:
aabb
That string does not match the expression.
Do you want another string for the same expression? n
Do you want to enter another regular expression? y
Please enter a regular expression:
(abc)+
Please enter a string:
abcabc
That string matches the expression.
Do you want another string for the same expression? n
Do you want to enter another regular expression? n
Thank you and goodbye.

You may assume that the regular expression and the string to be matched each fit on a single input line, so all you have to do is read one line at a time (e.g. use gets in C or getline in C++).
deterministic.) If there is any way at all to arrive at the final state after having reading the entire input string, the machine is said to accept the string. Otherwise, it rejects the string.

For example, the following NDFA accepts a string if and only if it is of the form $a (a | b) * b$:

![NDFA Diagram]

There are three states in this NDFA (labeled by numbers within the state nodes). The initial state is state 1, denoted by the caret attached to its node; the final state is state 3, denoted by the doubled circle. Note that situations can arise in which the machine has no available transition (for example, from the initial state, if the string begins with a `'b'`); in other situations, two transitions are available (for example, from state 2, if the next input character is a `'b'`). Also, note that a machine may accept a string even if some of the paths it follows lead to failure. For example, the above machine could execute two ways on the string "ab". It could move from state 1 to state 2 on the 'a', then loop at state 2 on the 'b', and not end up in the final state. Alternatively, it could move from state 1 to state 2 on the 'a', then move to state 3 on the 'b', leading to a final state. Since the second path does lead to a final state, the machine accepts "ab". (All that is necessary for a machine to accept a string is the existence of at least one transition path that ends up in the final state after processing the input string.)

### 12.6.1.3 Representing Regular Expressions with NDFAs

For every regular expression it is possible to construct an NDFA that accepts exactly the set of strings denoted by the regular expression. Here are the NDFAs for the basic regular expressions:

- # and alphanumeric characters ('a' in this case):
Starting with simple NDFAs like these, one can use a few basic constructions to build up more complex NDFAs to represent complicated regular expressions. For instance, we can construct an N DFA for $\alpha \mid \beta$ if we already have NDFAs for $\alpha$ and $\beta$. To do so, we add two new states and four new transitions, as illustrated below. The two boxes on the left represent the NDFAs that accept the sets of strings denoted by $\alpha$ and $\beta$, respectively. Only the initial and final states are shown; there may be a great many other states in each N DFA that are not shown. The N DFA on the right results from the NDFAs for $\alpha$ and $\beta$ by adding a new initial state with transitions (on reading no input) to the initial states of the original NDFAs, and a new final state with transitions (on reading no input) from the final states of the original NDFAs. The new N DFA can reach its final state while reading an input string if and only if one or the other of the original machines can reach its final state while reading the same input string. Thus, the new machine accepts exactly the set of strings denoted by the regular expression $\alpha \mid \beta$.

Similarly, here is the construction of an N DFA for $\alpha^*$ from one for $\alpha$:

A new state is added which becomes both the initial and the final state of the N DFA. There are empty string transitions from the new (initial) state to the previous initial state and from the previous final state to the new (final) state. The new machine will accept a string if and only if the original machine could make 0, 1 or more complete trips from its initial to its final state while reading that input. Thus, it accepts exactly the closure of the set of strings accepted by the first N DFA. (You might think that a simpler construction would achieve the same result, but be careful; it is easy to make a subtle mistake.)

Only two other constructions remain. The construction of an N DFA for $\alpha+$ from an N DFA for $\alpha$ is extremely similar to the construction of an N DFA for $\alpha^*$, above. The only difference is that the initial state of the new machine should remain the same as the initial state of the N DFA for $\alpha$. How about the construction of an N DFA for the concatenation $\alpha \beta$ (given NDFAs for $\alpha$ and
The initial state of the new machine should be the same as the initial state of the NDFA for $\alpha$; the final state should be the same as the final state of the NDFA for $\beta$; only a single transition (on no input) needs to be added, from the old final state of the $\alpha$ NDFA to the old initial state of the $\beta$ NDFA. (Try drawing diagrams of these two constructions to convince yourself that they will in fact accept the correct set of strings.)

12.6.1.4 Implementing NDFAs in C

The first stage of this project is to transform regular expressions into NDFAs. That is, you are to write a program that takes a regular expression as input and produces a computer representation of the corresponding NDFA. The exact form of the NDFA representation is left to you; clearly, it must be able to record the transitions from certain states to certain other states and the labels (alphanumeric character or empty string) on these transitions.

A recursive-descent parser is adequate to recognize regular expressions. In fact, since all lexical tokens are single characters, the scanner for regular expressions should be much simpler than the scanner for arithmetic expressions presented in the previous section. The parser should be quite similar to the arithmetic parser; the only subtlety involves the rule for $<RF>$. The semantic routines should combine NDFAs to produce new NDFAs according to the constructions above. For example, the semantic routine for the union of two NDFAs should take as its arguments (some representation of) the two component NDFAs and return as its value (some representation of) the resulting NDFA.

Thankfully, the constructions depicted above simplify the form that our NDFA can take. Notice that the constructions are designed so that no state will ever have more than two transitions out of it. Specifically, a state will have either no transitions out of it, a single transition upon reading an alphanumeric character, or one or two transitions without reading any input. This suggests that each state of a constructed NDFA can be described by three values, the first two describing if and where transitions from the state go and the third specifying the character (if any) upon which a transition can occur.

Whatever structure is chosen to represent NDFA states, the states can be allocated either dynamically (using malloc() in C or new in C++) or statically (by using an array of states, and keeping a count of how many states have been created so far). The static allocation strategy is simpler since it allows states to be named by their array indices (if states are allocated dynamically, they must be referred to by their addresses). Since the basic machines have two states and each time two machines are combined at most two new states are added, the number of states in the final NDFA is at most twice the number of characters in the original regular expression. If a limit is placed on the length of input, then, we know there will be a limit on the number of states created (twice the maximum input length).

Having such an array defined, the only information needed to fully describe an NDFA are the...
indices which correspond to the machine’s initial state and final state. (Keep this in mind when sending arguments to and returning values from the semantic routines.)

12.6.1.5 Simulating the Non-Deterministic Finite Automaton

The second stage of the project is to simulate the action of an NDFA on a particular input string. Since this is a type of graph-searching problem, an exhaustive search procedure involving back-tracking could be devised. Instead, we suggest a more clever strategy.

The idea is to read the input string (from left to right) only once, keeping track at all times of all possible states in which the NDFA could be when it reaches the current point in the string. Looking back at the first NDFA pictured in this project, the machine could be in either state 2 or state 3 after reading the input "abb". We will refer to this set of states as the current-state-set. At the beginning, the current-state-set will contain only the initial state. If the final state of the NDFA is in the current-state-set after reading the entire input, we know that the NDFA accepts the input string.

The simulation algorithm proceeds by considering each state from the current-state-set and determining what transitions are possible out of it. Since some transitions involve reading an input character and others do not, we must keep track separately of the states the NDFA could be in before reading the next input symbol and the states it could be in after reading the next input symbol. For this reason, a next-state-set must also be maintained. Suppose that state p is in the current-state-set with its only transition leading to state q upon reading an ‘a’:

\[ p \xrightarrow{a} q \]

When state p is processed, state q should be added to the next-state-set if the next input symbol is ‘a’. If the next input symbol is not ‘a’, however, then nothing should be added to the next-state-set.

Suppose instead that state r is in the current-state-set, and it has a transition to state s labeled with the empty string (i.e., a transition that can take place without reading any input):

\[ r \xrightarrow{} s \]

When state r is processed, state s should be added to the current-state-set. (Note: Adding a state to a set does nothing if the state is already in the set.)

When all states from the current-state-set have been processed (including any added as the result of non-reading transitions from a state already in the current-state-set), the
input pointer is advanced one character, the next-state-set replaces the current-state-set, the next-state-set becomes empty, and the processing of states in the current-state-set is repeated.

What kind of bookkeeping is needed when a current state is processed? It is tempting simply to remove the state from the current-state-set and to wait for this set to become empty before concluding that all current states have been processed. Unfortunately, this method does not work since there could be a sequence of transitions, none of which reads any input, that leads from a state back to itself:

In this case, the processing could loop endlessly without advancing the input pointer. Instead, you must maintain a separate processed-state-set, to which a current state is added once it has been processed. The input pointer should be advanced when the processed-state-set is equal to the current-state-set.

We present the entire simulation algorithm in C pseudo-code:

```c
current_state_set = {initial state};
input_pointer = {beginning of input string};

for(;;)
{
    processed_state_set = {empty set};
    next_state_set = {empty set};
    ...
}  
```
while (current_state_set != processed_state_set) do 
{
    state p = {a member of current_state_set that is not a 
    member of processed_state_set}
    for {each transition t out of state p} 
    {
        if {t is a transition out of p to state q while 
            reading symbol c or is a transition out of p to 
            state q on any input (i.e. the ? transition), 
            and input_pointer is not at the end of the 
            input string, and next input character is c} 
        {add q to next_state_set}
        if {t is a transition out of p to state q while reading 
            no input} 
        {add q to current_state_set}
    }
    {add p to processed_state_set}
}

if {input_pointer is at end of input string} break 
else 
{
    {advance input_pointer}
    current_state_set = next_state_set;
}

if {final_state is in current_state_set} 
{accept}
else 
{reject}

The final consideration is how to represent a set of states. You are free to choose whatever 
representation you wish. One possibility is an array set of booleans of the same length as the 
number of states (booleans may be defined as an enumeration type, or as the equivalent of a 
char). If consecutive numbers 0, 1, ... are available as the names of states, then set[i] can 
be made non-zero ("true") to indicate that state i is in the set; to indicate that it is not in the 
set, set[i] can be made zero ("false"). Using this implementation, we would need three such
arrays of chars to represent the three sets in the above algorithm. It is also possible to allocate
the booleans as part of the states themselves. But we recommend that you stick to the simplest
implementation, which is to have three arrays of booleans.

12.6.1.6 Optional Additions

- Most implementations of egrep also allow you to take advantage of the construction n?,
  where n represents a digit, and ? represents any character. This construction means, “accept
  any n characters.” The regular expression a5?b, for example, means accept all strings of
  length 7 that start with ‘a’ and end with ‘b’.

- Rather than implementing the suggested algorithm for simulating the operation of an NDFA,
  try using an exhaustive search procedure to perform the same simulation. See what effect
  this has on the running time of your program.

12.6.2 Extending the Arithmetic-Language Interpreter

12.6.2.1 A New and Improved Language

For this project, you will be modifying the code from the preceding section in order to build an
interpreter for a more complicated (and more useful) version of our arithmetic language. The
new version of our language will allow for three control structures: if–else statements, while
loops, and for loops.

The grammar for this enhanced language is as follows:

```
<program> ::= <stmt> .
<stmt> ::= <empty> | <asst> | <cond> | <loop> | <block>
<empty> ::= ;
<asst> ::= atom = <expr> ;
<cond> ::= if (<boolexpr>) <stmt> { else <stmt> }
<loop> ::= while (<boolexpr>) <stmt> |
          for (<asst> <boolexpr> ; <asst>) <stmt>
[block] ::= { {<stmt>}* }
<boolexpr> ::= <expr> <boolop> <expr>
<expr> ::= <term> { [+,-] <term> }*
<term> ::= <factor> { [*,/] <factor> }*
<factor> ::= atom | number | (<expr>) | -<factor>
<boolop> ::= == | != | < | <= | > | >=
```
The items in bold are tokens in our language. In particular, notice the difference between normal braces and bold braces. The bold braces in \(<block>\) indicate that a block is a series of zero or more statements surrounded by a left-brace character and a right-brace character. The normal braces surrounding the else-part of a \(<cond>\) statement indicate that the else-part is optional. Also notice that there are now four reserved words in our language — if, else, while, for. These words may not also serve as names for symbolic atoms.

The six boolean operators should operate much the same way as they do in C. Unlike C, however, you do not need to be concerned with assigning the value of a boolean expression to a symbolic atom since our grammar doesn’t allow it.

The three control structures are also quite similar to their C counterparts. One small difference exists within the for loop of our language. Notice that the second \(<asst>\) (the “iteration assignment”) indicates that there will be a semicolon just before the right parenthesis; this semicolon is omitted in C.

Our grammar is also ambiguous for a \(<cond>\) statement. For instance, it’s not clear to which if the else belongs in the following program:

\[
\begin{align*}
\text{if} & \quad (x < 3) \\
\text{if} & \quad (y < 5) \\
& \quad z = 4; \\
\text{else} & \quad z = 10; \\
\end{align*}
\]

We stipulate that an else belongs to the same \(<cond>\) statement as the most immediate if, just as is the case in C. So, for the above program, the else belongs to the \(<cond>\) statement which begins with if \((y < 5)\).

A while loop continues executing the \(<stmt>\) as long as its \(<boolexpr>\) is true. A for loop also continues executing as long as its \(<boolexpr>\) is true; the first \(<asst>\) is executed before the first iteration of the loop, and the second \(<asst>\) before each successive iteration.

A slight difference from the more simplistic version of our language is that a series of assignment statements must now be enclosed in braces to be a valid program. In addition, nonsensical programs like ; . and \{ \} are syntactically legal.

Here is a sample program which exhibits many of the features of our new language:

\[
\begin{align*}
\text{resultone} & = 1; \text{resulttwo} = 1; \\
a & = 3; \ b = 5; \\
\text{for} & \quad (\text{counter} = 0; \text{counter} < b; \text{counter} = \text{counter + 1};) \\
& \quad \text{resultone} = \text{resultone} \ast a;
\end{align*}
\]
\begin{verbatim}
c = 4; d = 4;
i = 0;
while (i != d)
{
  resulttwo = resulttwo * c;
i = i + 1;
}

if (resultone > resulttwo)
  max = resultone;
else
  max = resulttwo;
\end{verbatim}

This program determines the values of $a^b$ and $c^d$, and outputs the larger of the two. Notice that we’re interested in the value of the last assignment statement to be executed, which need not be the last assignment statement to appear in the program.

\subsection*{12.6.2.2 Implementing a Parse Tree}

The semantics for our improved language will clearly be more complicated than the semantics from the old version of the language. In the old version, all we had to do was process a series of assignment statements, performing the assignment indicated by a statement and continuing on to the next assignment statement. Now, however, we can’t simply go through the input sequentially since the language allows for control structures. In a \texttt{for} loop, for instance, we need to be able to “remember” the boolean test, the iterative assignment statement, and all of the statements within the body of the loop.

One way of remembering all of the information in a program is by constructing a parse tree when the program is passed through the parser. This parse tree will reflect all of the information contained in the program. Our interpreter could then reference any of the program’s statements and expressions by traversing the tree. In practice, we can have our semantic routines build this parse tree. After the tree is built, the program can be executed by “processing” the parse tree.

Certain tokens from our language will be worth remembering while others will not. For instance, since we know that every assignment statement ends with a semicolon, there is no need to store this token as one of the children of an \texttt{<asst>} node in our tree. Likewise, there is no need to store the token of type \texttt{T\_ASST}; the only important parts of an assignment statement are the symbolic atom being assigned and the arithmetic expression to which it’s being assigned. To fully
characterize an `<asst>` then, we need just two children branching from an `<asst>` node in the parse tree — one for the symbolic atom and one for the arithmetic expression.

What types of nodes will we need for our parse tree? We need just three of the token types — symbols, numbers, and operators (both arithmetic and boolean). None of the other token types are necessary since they are only used to supply correct syntax, not to add any useful information to the program. We’re also going to need additional types of nodes in order to know which constructs from the language are being used in the program. For this purpose, we’ll need five more types of nodes (for assignment statements, conditional statements, for loops, while loops and blocks). Notice that there is no need to have a node for a `<stmt>` since our grammar tells us that a statement must be uniquely characterized by one of these five types of nodes.

We can now define our node types in C:

```c
typedef enum {
    N_SYM, /* symbol */
    N_NUM, /* number */
    N_UOP, /* unary operator */
    N_BINOP, /* binary operator */
    N_ASST, /* assignment stmt */
    N_COND, /* if ... */
    N_WHILE, /* while loop */
    N_FOR, /* for loop */
    N_BLOCK /* block of stmts */
} NODE_T;
```

We’ve made a distinction between unary and binary operators here since a unary-operator node will have just one child. (The only unary operator currently supported by our language is the negation operator.)

The nodes representing symbols and numbers clearly will not have any children. After all, they are terminals of our grammar. The node for a unary operator will have a single child while the node for a binary operator will have two children. As explained above, the node for an assignment statement will also have two children. A conditional-statement node, however, will require three children; the third child will represent the optional `else`-part of the conditional statement. The node for a `while`-loop needs two children — one for the boolean test and one for the loop body’s statement. The `for`-loop node requires four children — three to take care of the boolean test and the counter’s initialization and iteration, and one to take care of the loop body’s statement. It seems that a block would need an indeterminate number of children since it can consist of any number of statements. However, there is a way to implement the block node with only two children. The first child of a block node will be a statement, and the second child will be another block node. In this way, we can chain together as many block nodes as necessary in order to represent a series of
CHAPTER 12. LEXICAL ANALYSIS AND PARSING

statements; the last block in the chain will have no second child (i.e., a NULL pointer).

Any node in our tree, then, will have a maximum of four children. With this in mind, we can
construct a data structure to represent any node:

typedef struct tnode {
    NODE_T type; /* type of node */
    char *pname; /* print name */
        /* (for symbolic atoms) */
    struct tnode *nextatom; /* table link */
        /* (for symbolic atoms) */
    int value; /* value (for numbers, */
        /* symbols, operators) */
    int inited; /* init flag (for symbolic */
        /* atoms) */
    struct tnode c1, c2, c3, c4; /* pointers to */
        /* children (max of 4) */
} tnode, *tnodeptr;

Most of these fields should be familiar to you from our lexeme definitions of the preceding section.
Three fields (pname, nextatom, inited) are used only for symbolic atoms. The value field
can be used for numbers, symbolic atoms, or operators. To deal with the additional operators in
our new language, we need an extension of our definition of OP_TYPE:

typedef enum { OP_NEG, OP_PLUS, OP_MINUS, OP_MULT, OP_DIV,
        OP_EQ, OP_NEQ, OP_LESS, OP_LESSEQ, OP_GREATER,
        OP_GREATEREQ } OP_TYPE;

Finally, the pointer fields (c1, c2, c3, c4) will be used according to how many children a particu-
lar node has. These fields should be used in order — c1 should be used before c2, c2 before c3,
and c3 before c4. For instance, a node of type N_ASST would use c1 and c2; c1 would point to
a node of type N_SYM, and c2 would point to a node of type N_UOP or N_BINOP.

Consider the following program:

\{
    a = 10;
    b = 1;
    while (a > b)
        b = b * 2;
\}

Here is the complete parse tree which should be constructed from this program (only the ap-
propriate fields for each node are shown):
Aside from the numbers, symbolic atoms, and the final block node, each of the nodes in this parse tree has precisely two children. Notice that each symbolic atom is stored only once. Each time a given symbolic atom needs to be referenced, a pointer will point to the single existing copy of that symbolic atom. As in our original interpreter, the atoms are linked together in a linked list via the `nextatom` field, and the `atomlist` pointer points to the head of this linked list. The
value and initied fields for symbolic atoms will change as the parse tree is processed, but when we first create the parse tree, we want to turn off the initied flag for each atom.

12.6.2.3 Modifying the Old Interpreter

Our original interpreter was broken up into three parts: the lexical analyzer, the parser, and the semantics. Each of these parts will have to be changed in order to support the new additions to our language and the implementation of a parse tree. In addition, a parse-tree processor will have to be written from scratch to handle the parse tree created by the three aforementioned parts of the interpreter. Let’s see how the new interpreter should function, breaking it down into its four component parts:

12.6.2.3.1 The Lexical Analyzer. The scanner must be expanded to reflect the new tokens and reserved words of our language. Some of the new tokens won’t be recognizable from a single character. For instance, if we read in the ‘<’ character, we can’t be sure whether we’re encountering the less-than operator or the less-than-or-equal operator; to figure out which type of token is being scanned, the next character must be checked to see if it is an equal sign. Another important consideration is being able to distinguish the language’s reserved words from ordinary symbolic atoms. If the string "while" is read in (followed by some non-alphabetic character, of course), the scanner should return a token that reflects the fact that the syntactical beginning of a while loop has been encountered. A symbolic-atom token should not be returned in this case.

Now that we’re using a parse tree, our tokens don’t have to store as much information as before. In fact, we can simplify our definition of the type lexeme as follows:

```c
typedef struct {
    LEXTYPE     type;
    char        pname[SYMBOL_SIZE];
    int         value;
} lexeme, *lexptr;
```

You will have to change the definition of LEXTYPE in order to cover the additional types of tokens allowed by our language. The value field of lexeme should be used only when returning a numeric token or an operator token. The pname field is used only when returning a symbolic-atom token. Unlike our original scanner, the new scanner will not need to call lookup since we won’t worry about the value and initied fields of symbolic-atom nodes until the parse tree has been built. These fields will be used when we process the parse tree; for this reason, initied is not even part of our lexeme structure, and the value field need not be used when returning symbolic-atom tokens.
12.6. PARSING PROJECTS

12.6.2.3.2 The Parser. Our new parser will still be a recursive-descent parser, but since the rules of our grammar have changed, the functions that make up the parser will also have to change to correspond to the new grammar rules. The goal of our parser is to build up a parse tree consisting of nodes of type tnode. (Recall that our original parser simply returned an integer value.) With this in mind, our parse routines should be defined to have tnodeptr as their return type. Each function will return a pointer to a tree structure which represents how the corresponding rule is parsed.

To help clarify the form of your new parser, we offer two of the parsing functions, corresponding to the rules for <stmt> and <asst>:

```c
/* stmt(): handle stmt rule */
static tnodeptr
stmt(void)
{
    switch(getlex()->type)
    {
    case T_SEMICOLON: /* skip empty */
        return NULL; /* statements */
    case T_SYM:
        { ungetlex(); /* don’t lose */
        return asst(); /* symbolic atom */
        }
    case T_IF:
        return cond();
    case T_WHILE:
        return while_loop();
    case T_FOR:
        return for_loop();
    case T_LBRACE:
        return block();
    default:
        error("invalid statement");
    }
    }
```
/* asst(): handle asst rule */
static tnodeptr
asst(void)
{
    tnodeptr lhs, rhs; /* left, right hand side of */
    /* assignment stmt */

    lhs = sym_sem(getlex()->pname);
    if (getlex()->type != T_ASST)
        error("invalid assignment statement");
    rhs = expr();
    if (getlex()->type != T_SEMICOLON)
        error("missing semicolon in "
        "assignment statement");

    return asst_sem(lhs, rhs);
}

Several of the function calls from within stmt and asst indicate what names should be given to some of the additional parsing functions which you will be writing. Also, notice that there are calls to two semantic routines (sym_sem and asst_sem) within asst. Such semantic routines will create the actual tree structure (and return a pointer of type tnodeptr). For instance, asst_sem will return a tnodeptr pointer which points to a node of type N_ASST that has the assignment statement's left-hand side and right-hand side as its two children, respectively.

12.6.2.3 Semantics. As just explained, the semantics will build up the actual parse tree. To see how this works, let's look at how one of the actual routines would be written. Consider the function asst_sem (which is called from the parse routine asst):
12.6. PARSING PROJECTS

/* asst_sem(): glue lhs and rhs of = together */

```c
tnodeptr
asst_sem(tnodeptr lhs, tnodeptr rhs)
{
    tnodeptr a = nalloc();

    TYPE(a) = N ASST;
    CHILD1(a) = lhs;
    CHILD2(a) = rhs;
    CHILD3(a) = CHILD4(a) = NULL;

    return a;
}
```

Notice how we allocate memory for the pointer variable `a` in the first line of the function. We define `nalloc()` to be a macro that uses `safe_malloc`(as defined in our original semantic routines):

```c
/* allocate memory for a tnode */
#define nalloc() (tnodeptr)(safe_malloc(sizeof(tnode)))
```

After space has been allocated for the node structure, we assign to the node the appropriate type — N ASST. Notice that another macro is being used here, rather than cryptic pointer notation (e.g., `a->type = N ASST;`). Here is a list of macros for making your semantic routines more readable:

```c
/* macros to access fields of a tree node */
#define TYPE(n) (n)->type /* type */
#define PNAME(n) (n)->pname /* print name */
#define NEXTATOM(n) (n)->nextatom /* next symbol in symbol */
#define VALUE(n) (n)->value /* for numbers, symbols, */
                 /* list */
#define INITED(n) (n)->inited /* for symbolic atoms */
                 /* only */
#define CHILD1(n) (n)->c1 /* the four child */
#define CHILD2(n) (n)->c2 /* pointers */
#define CHILD3(n) (n)->c3
#define CHILD4(n) (n)->c4
```

These macros are used in `asst_sem` to point the first-child pointer at the tree structure which represents the left-hand side of the assignment statement and the second-child pointer at the tree.
structure which represents the right-hand side. Since only two child pointers are needed, the other child pointers are initialized to NULL. (As we’ll see later, this is necessary so that memory can eventually be de-allocated properly.) None of the other tnode fields are applicable and, thus, are not mentioned in asst_sem.

Since your semantic routines are responsible for creating unique copies of symbolic-atom nodes, you will find it useful to use a modified version of the function lookup from our original interpreter. Remember that we are now interested in searching for a tnodeptr rather than a lexptr.

12.6.2.3.4 The Parse-Tree Processor. Once the scanner, parser, and semantics have been completed, it should be possible to create a parse tree for any syntactically correct program. To execute such a program, you need to write a parse-tree processor. Since we’re interested in the value of the last assignment statement executed within the program, the ultimate return type of this processor should be an int.

Just as the parser has a routine for each non-terminal of the grammar, your parse-tree processor should have a routine corresponding to each of the nine types of tree nodes (e.g., process_asst, process_while, process_binop, process_symbol, etc.). The routine for processing a while loop, for instance, should take a pointer to a node of type N_WHILE as its argument; the routine should continuously check if the associated boolean expression is true and, if so, execute the associated loop statement. Notice that these two operations (checking the boolean expression and performing the loop statement) can actually be done by calling upon two other processing routines.

As a complete example of a processing routine, consider how we would write process_asst:

```c
static int
process_asst(tnodeptr t)
{
    VALUE(CHILD1(t)) = process(CHILD2(t));
    INITED(CHILD1(t)) = TRUE;
    return (VALUE(CHILD1(t)));
}
```

This routine first sets the value of the symbolic atom on the left-hand side of the assignment statement (pointed to by CHILD1(t)) to the value on the right-hand side of the assignment statement (pointed to by CHILD2(t)). Then, it signals that this symbolic atom has been assigned, and returns the value just assigned. The function process is assumed to be a general routine (which you will have to write) that takes any tree-node pointer and calls the appropriate process routine, depending on the node’s type. After all, we don’t know whether CHILD2(t) is a symbol, atom, or binary operator; rather than considering each case, we let the function process do this check.
A little extra work needs to be done to have our processor remember the value of the last executed assignment statement. For instance, if the last statement in the program is a while loop in which no assignment statement is executed (i.e., the boolean test fails on the first pass), we want to return the value of the last assignment statement executed before invoking the while loop. Also, note that you should not consider the assignment statements occurring within the parentheses of a for loop as candidates for the final return value of the processor. A single temporary variable (perhaps local to the processing routines) should be sufficient to implement a method for remembering the appropriate return value. Figure out how this temporary variable should be used throughout your processing routines. (Notice that there is no mention of this local variable within process_asst; why?)

To invoke the parse-tree processor from main(), the line

```c
value = program();
```

should be changed to

```c
parse_tree = program();
value = process_prog(parse_tree);
```

where `parse_tree` is a variable of type `tnodeptr` and `process_prog` is the name of the main processing routine.

### 12.6.2.4 De-Allocating Memory

The building of a parse tree requires the allocation of quite a bit of memory by the semantic routines. After the parse tree has been processed, the parse tree — and the memory used to store it — is no longer needed. How about just declaring a function for freeing up the allocated memory as follows:

```c
static void
delete(tnodeptr t)
{
    if (t != NULL)
    {
        delete(CHILD1(t));
        delete(CHILD2(t));
        delete(CHILD3(t));
        delete(CHILD4(t));
        free(t);
    }
}
```
Unfortunately, the special treatment given to symbolic atoms causes this function to run into trouble. Since symbolic atoms can be pointed to by more than one tree node, a problem could arise when we attempt to delete a symbolic-atom node more than once (i.e., after the node is initially free’d, its pointer fields are free to be filled with some other information before an attempt is made to delete the node again).

To circumvent this problem, the following algorithm can be used:

- Delete all non-symbolic nodes from the parse tree (using a routine like delete).
- Delete all symbolic nodes (traversing the linked list of symbolic atoms and de-allocating memory as you go along).

Once you’ve implemented this algorithm for memory de-allocation, you’ll need to have a call to your de-allocation routine from main() just after the parse tree has been processed. You should call this routine with parse_tree as its argument (see the modification to main() suggested above).

### 12.6.2.5 Optional Additions

- As the interpreter is set up, any error (syntactical or otherwise) results in a call to exit. It would be more reasonable to ask for another program as input when certain errors, called recoverable errors, occur. For instance, there is no reason that the interpreter should exit when input contains a syntax error. On the other hand, it is reasonable to exit when a fatal error occurs. In our interpreter, lack of memory available for allocation should be considered a fatal error; if there’s no memory available, there is no way to build a parse tree! Try to distinguish between recoverable errors and fatal errors within your program; the program should prompt for more input in the former case and exit in the latter case.

- The interpreter handles only very basic boolean expressions. It would be more useful if we could handle the logical operators and, or, and not. We could then have boolean expressions like ((a > 0) and (a < 10)) and (not (a == 3) or (b != 6)). Try to figure out a way to incorporate these logical operators into the grammar. (It’s probably easiest to follow the example of the arithmetic operators.) Remember that and takes precedence over or; also, notice that not is a unary operator while and and or are binary operators. Once you’ve incorporated these operators into the grammar, modify the code for your interpreter appropriately.

- Add more control structures to the language. An easy one to implement is a do-while loop (analogous to C’s do-while construct, or Pascal’s repeat-until construct). Im-
implementing a version of C’s switch-case construct is much more difficult. Think of some other possibilities, like branching statements.

- Allow for comments within programs — set off by /* and */, as in C. This should be a rather easy modification; only the scanner has to be altered.

- Allow for input from external text files.

12.6.3 Follow-up to the Arithmetic Interpreter: Compilation

12.6.3.1 Introduction to Assembly Language

The goal of this project is to expand upon the work done for the arithmetic interpreter in order to build a compiler (code generator) for our arithmetic language. Given a program in the language, we would like to generate assembly-language code which, if executed, would behave just like the program does.

Only the semantic routines in the file semantics.c will have to be changed to build our compiler (alternatively, if you have implemented the parse-tree in Section 12.6.2, only the parse-tree processor need be changed, with a small exception). Our first goal is to generate code for programs consisting of simple sequences of assignment statements (i.e., containing none of the control structures introduced in Section 12.6.2). After this is completed, it will be left as an exercise to extend the code generator so that it works for all possible programs.

In order to build a compiler, we first need to have some understanding of assembly language. Unfortunately, there are many, many versions of assembly language; the type of assembly language implemented on a given computer depends on that computer’s architecture. We will shortly introduce a bare-bones assembly language which will be used in conjunction with our compiler. After all, we’re not really interested in how the generated code would be executed; rather, we’re interested in how this code is generated. (Note: If you’d like to try executing generated code after you’ve finished this project, you should modify the compiler so that the assembly-language commands it generates correspond to the commands supported by your machine.)

12.6.3.1.1 Registers. Registers are common to all versions of assembly language. A register is simply a reserved location which the computer uses to store information. Many of the instructions in assembly language operate on these registers. (Instructions which use registers are much quicker than those that deal with information stored at other memory locations.)

There are two types of registers: data registers and address registers. As their names suggest, data registers are used to store data (e.g., an integer, a character value, etc.) and address registers are used to store addresses. An address register can be thought of as a pointer since it “points” to the object located at its stored address. The number of registers supported depends on the particular
assembly language being used. Our assembly language will have eight data registers named \texttt{d0}, \texttt{d1}, \texttt{d2}, \texttt{d3}, \texttt{d4}, \texttt{d5}, \texttt{d6}, and \texttt{d7}. We also have eight address registers named \texttt{a0}, \texttt{a1}, \texttt{a2}, \texttt{a3}, \texttt{a4}, \texttt{a5}, \texttt{a6}, and \texttt{a7}.

Since we only deal with integers in our arithmetic language, we will use only one type of data, an integer stored in two memory locations. A memory location is called a byte, and is eight bits in size (a bit is a binary digit). Two bytes hold a 16 bit integer, which can range in value from -32768 through +32767.

We will use one of our address registers, \texttt{a5}, to point at a vector of integers whose elements will be the variables defined within an arithmetic-language program. The memory location of each of the variables in the vector, then, can be referenced as some offset from the address within \texttt{a5}. This offset indicates where in memory the value of a variable is stored. The pointer \texttt{a5} will actually remain constant throughout (i.e., it will always point to the same location in memory — the location corresponding to the first variable vector element).

In order to perform arithmetic calculations, we'll need to store results in temporary locations. For instance, in determining the value of the expression \((12+3) \times (4+6)\), we need to first take the sum of 12 and 3 and set the result aside so that we can later multiply it with the sum of 4 and 6. We will use a stack to hold temporary values. The top element of the stack we will put in \texttt{d0}, so we can get at it easily. The second element of the stack will be pointed at by the address register \texttt{a7}, so we can have the stack be as long as necessary, limited only by the size of memory, and not the number of registers.

To calculate \((12+3)\) for example, we push 12 into the stack, then push 3 into the stack, and then perform a “stack-add” operation that pops the top two elements from the stack, adds them, and pushes the sum back onto the stack. The stack now has one element in it. If we then compute \((4+5)\) the same way, the stack will have two elements, the two sums. We can then do a “stack-multiply” operation to get the product we want at the top of the stack (in \texttt{d0}).

Since the stack (at least in theory) can hold an arbitrary number of values, it solves all of our temporary storage problems. The pointer \texttt{a7}, unlike \texttt{a5}, will change. Because \texttt{a7} points to the “rest” of the stack, the address at which it points will change according to whether a value is added (pushed) on top of the stack or removed (popped) off the top of the stack.

### 12.6.3.1.2 Instructions

Before getting into the specifics of how registers are manipulated by our compiler, let’s first examine some of the basic instructions of our assembly language. There are six arithmetic operations:
Four of the operations are binary. In each case, the operation is applied to the values of `<src>` and `<dst>` with the result replacing the value of `<dst>`. The other two operations, `neg.w` and `ext.l`, are unary operators. `neg.w` negates the value which is in `<dst>`. `ext.l` takes a 16 bit integer in `<dst>` and converts it to a 32 bit integer in `<dst>` (it “extends” the 16 bits to 32 bits).

The `.w` operations operate on 16 bit integers and produce 16 bit results. The `muls` takes 16 bit inputs and produces a 32 bit integer as an output. However, to convert a 32 bit integer to a 16 bit integer, you do not need to execute any instructions at all if the integer is in a register; you merely use the 32 bit integer as if it were a 16 bit integer. The `divs` instruction takes a 16 bit `<src>` and a 32 bit `<dst>` and produces a 16 bit quotient in `<dst>` (remainders are thrown out). We will need the `ext.l` instruction to take a 16 bit integer in `<dst>` and prepare it for use as a 32 bit integer value in the `<dst>` of a `divs`.

Only one more command is needed to generate code for the simplified version of our arithmetic language (i.e., the one without control structures). This command allows assignment of values:

```
move.w <src>, <dst>
```

The value of `<dst>` is replaced by the value of `<src>` (without the value of `<src>` being affected).

### 12.6.3.1.3 Addressing Modes.

What can be used as `<src>` or `<dst>` in the seven instructions described above? For our purposes, these arguments can take on any of the following five “addressing modes”:

1. **Data Register Direct**
   - The value is contained within a data register.
   - ex: `add.w d0, d0`

2. **Immediate**
   - The value is a specified number (denoted by a leading # character). (Note: The `<dst>` operand can’t be immediate since the value of a number can’t be changed.)
   - ex: `move.w #3, d0`
3. **Address Register Indirect with Displacement**

   The value is in a location which is an offset from the location pointed to by an address register. The address register is enclosed within parentheses to indicate indirect addressing, and the offset is specified by a (possibly negative) number preceding the left parenthesis.

   ex: `move.w 2(a5), d0`

4. **Address Register Indirect with Post-Increment**

   The value is in a location pointed to by an address register. After the command is executed, the address register is incremented by the size of the value; e.g., if the value is a two byte integer, the address register is changed to point to the integer just after that to which it had been pointing.

   ex: `move.w (a7)+, d0`

5. **Address Register Indirect with Pre-Decrement**

   Before the command is executed, the address pointer is decremented by the size of the value to be accessed (e.g. by 2 if the value is a 2 byte integer). The command is then executed, using the new location at which the address register points.

   ex: `move.w d0, -(a7)`

The last three addressing modes require a little extra explanation since they are used frequently in conjunction with vectors and stacks. Let’s go through each of the given examples for the last three addressing modes.

For address register indirect with displacement, `2(a5)` indicates that we’re interested in the value that occupies the memory location which is offset by two memory locations (bytes) from the location pointed to by `a5`. If we think of `a5` as pointing to the first element of a vector, `2(a5)` just references the second element on the vector, remembering that each element takes 2 memory locations:

```
0 (a5)  8
2 (a5) 17
4 (a5) -4
6 (a5) 12
...    
```

The example command, then, moves the value of the second vector element, 17, into `d0`.

The last two addressing modes correspond to the stack operations of popping and pushing, respectively. When stacks are built up in memory, the higher elements (i.e., those closer to the top)
are actually at lower memory locations in the machine. This means that the topmost elements of the stack occupy the lower-numbered memory locations in the stack.

Remember that \( d_0 \) holds the topmost element of our stack, and \( a_7 \) points at the second element of the stack. To pop the stack, we must copy the second element into \( d_0 \), and increment \( a_7 \) by 2, so it now points at the former third element. \((a_7) +\) indicates that we first want the value pointed at by \( a_7 \). After the command is executed with this value, the \( a_7 \) pointer is incremented by the length of the value moved, in this case 2, since two bytes were moved. In effect, the example command pops the stack by moving the second stack element into \( d_0 \) and incrementing \( a_7 \) by 2:

\[
\begin{array}{c|c|c}
\text{BEFORE} & \text{AFTER} \\
\hline
\vdots & \vdots & \vdots \\
-6 & 11 & -6 \\
11 & -13 & \vdots \\
a_7 & \vdots & d_0 \\
\end{array}
\]

On the other hand, \(- (a_7)\) decrements the \( a_7 \) pointer before the command is executed by the size of the value being accessed, in our case, 2. The new location at which \( a_7 \) points is then used when the command is executed. Thus, our example command effectively pushes the value from \( d_0 \) onto the rest of the stack:

\[
\begin{array}{c|c|c}
\text{BEFORE} & \text{AFTER} \\
\hline
\vdots & \vdots & \vdots \\
-8 & 4 & \vdots \\
33 & -8 & \vdots \\
a_7 & \vdots & a_7 \\
\vdots & d_0 & \vdots \\
\end{array}
\]

This command is not all that is required to push a value into the stack: it merely frees \( d_0 \), and must be followed by some command that copies the value into \( d_0 \).

What if the stack is empty when we want to push a value into it. Then \( d_0 \) has no meaningful value, and we should not do the \texttt{move.w d0, -(a7)} that prepares \( d0 \) to receive the value. In the code we will explain below, we maintain a variable named \texttt{save} which is true if and only if the stack is non-empty, i.e. if \( d0 \) must be saved when pushing into the stack.
12.6.3.2 Compiler Semantics

We now present our compiler semantics routines:

```c
/* Compiler semantics routines */
** File: compiler-semantics.c
** Project: Arithmetic Language Compiler */

/* The code generator uses only a stack, whose top is in d0 and whose second topmost element is pointed at by a7. All arithmetic functions ignore their args, the actual args being on the stack, and return 0, the actual return value being left on top of the stack (in d0). Variables are stored in a vector whose base is pointed at by a5. */

#include "data.h"
#include "extern.h"
#define BYTES_PER_INT 2

static int save; /* True if there is something in d0 worth saving; i.e. if the stack is non-empty. */
static int symatomcnt; /* Number of symbolic atoms seen so far. */

As we said in a previous section, we need `save` to know when the stack is empty, so we can avoid the `move.w d0,-(a7)` instruction when pushing into the stack in this case. We need `symatomcnt` to assign offsets to variables in the variable vector. We will store the offset in the value field of the variable’s symbolic atom. Thus when we define a new variable, we will set its offset to `BYTES_PER_INT` times `symatomcnt`, before we increment `symatomcnt`.

Continuing, we see that the semantic routines for arithmetic operations merely output code that manipulates the stack. The arguments to these semantic routines are not used, nor is the return value, but we must return something to keep the parser happy, so we just return a meaningless zero.
/ * init_sem -- initialize variables for code generation
 */
 void init_sem(void)
 {
  save = FALSE;
  symatomcnt = 0;
 }

/*
 * sum_sem -- generate code for addition
 */
 int sum_sem(int v1, int v2)
 {
  printf("add.w (a7)+, d0\n");
  return 0;
 }

/*
 * diff_sem -- generate code for difference
 */
 int diff_sem(int v1, int v2)
 {
  /* Awkward because subtrahend starts off in d0, 
   ** diminuend on stack. 
   */
  printf("sub.w (a7)+, d0\n");
  printf("neg.w d0\n");
  return 0;
 }
/ * prod_sem -- generate code for product
 * /
    int
prod_sem(int v1, int v2)
{
    printf("muls (a7)+, d0\n");
    return 0;
}

/*
* div_sem -- generate code for division
*/
    int
div_sem(int v1, int v2)
{
    printf("move.w (a7)+, d1\n");
    printf("ext.l d1\n");
    printf("divs d0, d1\n");
    printf("move.w d1, d0\n");
    return 0;
}

/*
* sym_sem -- generate code for symbolic atom
*/
    int
sym_sem(lexptr l)
{
    if (save)
        printf("move.w d0, -(a7)\n");
    printf("move.w %d(a5), d0\n", 1->value);
    save = TRUE;
    return 0;
}
/ * num_sem -- generate code for numeric atom
 */

int
num_sem(lexptr l)
{
    if (save)
        printf("move.w d0, -(a7)\n");
    printf("move.w #%d, d0\n", l->value);
    save = TRUE;
    return 0;
}

/ *
 * neg_sem -- generate code for negation
 */

int
neg_sem(int v)
{
    printf("neg.w d0\n");
    return 0;
}

Note that the assignment operation below is the only place where the stack may be emptied. Conversely, the symbol and number operations above are the only place an empty stack becomes non-empty. The sym_init function is identical to that for the interpreter, except that the value field is set to the variable’s offset. The lookup function is actually identical to that for the interpreter:
/* 
* asst_sem -- generate code for assignment statement 
* /
int
asst_sem(lexptr l, int v)
{
    printf("move.w d0, %d(a5)\n", l->value);
    l->inited = TRUE;
    save = FALSE;
    return 0;
}

/* 
* sym_init -- initializes a new symbolic atom 
* (invoked from lookup) 
*/
static void
syminit (lexptr l, char *p)
{
    l->type = T_SYM;
    l->pname = (char *) safe_malloc
        (sizeof(char) * (strlen(p) + 1));
    if (l->pname == NULL)
        error("out of memory");
    strcpy(l->pname, p);
    l->inited = FALSE; /* no value for this atom yet */
    l->value = BYTES_PER_INT * symatomcnt++;
    l->nextatom = atomlist; /* insert atom into list */
    atomlist = l;
}
/* lookup -- looks for symbolic atom in symbol table
  * (a linked list). If it’s not there, adds it and
  * returns address of new atom; if it is there,
  * returns address of atom in the list.
*/

lexptr
lookup(char *p, lexptr newatom)
{
    lexptr theLex;

    for ( theLex = atomlist;
        theLex != NULL;
        theLex = theLex->nextatom
    )
        if (!strcmp(p, theLex->pname))
        {
            /* If it’s in the list, release the
             ** the space allocated, and return the
             ** address of the existing atom.
             */
            free(newatom);
            return theLex;
        }

    /* if it’s not in the list, initialize the new atom */
    syminit(newatom, p);
    return newatom;
}

12.6.3.3 An Optimization

Our compiler is not very optimal when it comes to pushing variable values and constants into the
stack. If one stops to think about it, there is no reason to ever push such things into the stack when
the compiled code is running. (This is in part because our language cannot change variable values
except when an assignment statement empties our stack.)

For example, 3+4 would be compiled by our compiler to:
CHAPTER 12. LEXICAL ANALYSIS AND PARSING

move.w #3, d0
move.w d0, -(a7)
move.w #4, d0
add.w (a7)+, d0

but it could be compiled to
move.w #3, d0
add.w #4, d0

The way to do this is to keep a stack at compile time that tells where every value in the conceptual arithmetic stack is. This permits elements of the conceptual stack to be somewhere else than the real stack: in particular, they can be constants or variable values. Thus when the semantic add operation is invoked, it would find that the compile time stack said the top conceptual stack element was #4 and the second conceptual stack element was #3, and would compile code appropriately. For that matter, it might even just add 4 to 3 at compile time and push #7 in the compile time stack, never outputing any assembly code at all.

The elements of your compile time stack will have to be structures with two fields. The first will be a type field to indicate whether the value is a constant, variable value, or a real temporary value actually in the stack at runtime. The second will be a “value” field to hold the value of a constant or the offset of a variable.

The binary operator semantics will become much more complicated because they have to handle roughly four cases: both arguments in the real stack, the first in the real stack and the second a constant or variable value, the second in the real stack and the first a constant or variable value, or both arguments not in the real stack. The last case could be further divided into two cases: both arguments are constants, or at least one is a variable. The trick here is to get the different operator semantic functions to share code to handle these cases, as much as possible.

12.6.3.4 Comparisons and Branches in Assembly Language

In order to extend the compiler to handle the control structures of Section 12.6.2, you will need some further assembly-language background. Since each of the control structures requires some type of comparison test, you’ll need a suitable comparison instruction in assembly language. In addition, you’ll need branching instructions to indicate the set of instructions that should be executed given the result of a comparison test. For instance, in a while loop, once the comparison test fails you’ll want to jump past the loop instructions; the final instruction within the loop instructions will be a branch back to the comparison test (so that the loop can be repeated, if necessary).

There is a single comparison instruction in our version of assembly language:

```
cmp.w <src>, <dst>
```
By itself, a `cmp.w` instruction is not meaningful. However, when followed by one of the following six branching instructions, the instruction allows us to branch to different parts of our assembly-code depending on the relative values of `<src>` and `<dst>`:

- `beq @<label>` (branches to `<label>` if `<dst> == <src>`)  
- `bne @<label>` (branches to `<label>` if `<dst> != <src>`)  
- `blt @<label>` (branches to `<label>` if `<dst> < <src>`)  
- `ble @<label>` (branches to `<label>` if `<dst> <= <src>`)  
- `bgt @<label>` (branches to `<label>` if `<dst> > <src>`)  
- `bge @<label>` (branches to `<label>` if `<dst> >= <src>`)  

If the condition associated with a given branching instruction is not satisfied, execution of the assembly code simply falls through to the next instruction.

In addition to these six branching instructions, there is another instruction, `bra`, for unconditional branching (i.e., no `cmp.w` instruction is needed):

- `bra @<label>` (branches to `<label>`)  

A `label` in assembly language can be placed before any instruction and is used as an argument to a branching instruction in order to continue execution of the assembly code at the labeled instruction. To indicate that a string is a label, a colon is placed after it. (Note: Each label in an assembly language program must be distinct.)

Consider how we might implement a `while` loop in assembly language through the use of a comparison test, branching instructions, and labels:

```
1:  
   .  
   .  
   cmp.w <src>, <dst>  
   b?? @2  
   .  
   .  
   <instructions in the loop's body>  
   .  
   .  
   bra @1  
2:  
```

After we’re able to process the nodes to be compared, the comparison test is performed. Then, a branching instruction (any of the first six described above) is used to branch to label 2 if the comparison test **fails**. For instance, if the `while` loop continues only if `<src>` and `<dst>` are equal,
the appropriate branching instruction would be \texttt{bne}. If the comparison succeeds, the branching instruction is passed over and the instructions in the loop’s body are executed. Finally, there is a branch back to the beginning of the \texttt{while} loop, labeled by 1.

Note that the \texttt{cmp.w} instruction leaves its arguments unaffected.

You now know enough about assembly language to extend the compiler to cover the syntax of the extended language in Section 12.6.2. You can in fact do this without building a parse tree, but the functions which parse \texttt{if-else}, \texttt{while}, and \texttt{for} will have to call several semantic routines. For example, \texttt{if} must somehow output a branch immediately after parsing its \texttt{<boolexp>}, then another branch and a label after parsing its first \texttt{<stmt>}, and lastly a label after parsing its second (else) \texttt{<stmt>}.

12.6.3.5 Optional Additions

- Since our version of assembly language supports just one data register, our generated code is highly dependent on the stack for temporary storage. If more data registers were supported, there would be fewer pops and pushes and, thus, fewer lines in the generated code. Try modifying the compiler so that it supports some multiple number of data registers — \texttt{d0}, \texttt{d1}, \texttt{d2}, etc. With multiple data registers, the stack should only be used when all of the data registers are full.

- When programming in assembly language, it is useful to heavily comment your code (sometimes even line-by-line). Comments are usually put after an assembly instruction and are preceded by a semicolon. Try to make your code generator also generate comments (although they need not be line-by-line). To do this, you should use actual lines of the arithmetic-language program to comment the corresponding blocks of assembly-language code to which they are equivalent.
Index

#', 32, 40–42
', 45
*, 44
* naming convention, 29
+, 44
-, 44
.asciiiz, 225
/, 45
/=, 43
<, 43
<=, 43
=, 43
>, 43
>=, 43
1’s complement, 206
1+, 44
1-, 44
2’s complement, 206
A* algorithm, 163
abstract class, 360
abstraction, 52
    lambda, 69
    procedural, 66
accept, 425, 426
add.asm, 218
add.asm (complete listing), 221
add2.asm, 223
add2.asm (complete listing), 224, 262
addition, see +
address, 309
address register, 445
address register indirect
    addressing mode, 448
addressing modes, 447
ako, 114
ako-get, 117
ako-network, 114
ako?, 116, 118
alignment restrictions, 254, 255
and, 48
and (\wedge), 171
animals project, 130
append, 46
apply, 42
aref, 38
arithmetic language, 393
array, 24
array-dimensions, 25, 38
ASCII, 210
assembly, 217
assert, 52
assignment statement, 393
assoc, 46
atoi-1.asm (complete listing), 268
atoi-4.asm (complete listing), 270
atom, 2, 393, 394
atom, 43
base class, 360
base-case rule, 61
binding
concatenation, 422, 423
cond, 33, 47
conjunctive normal form, 177
cons, 20, 36
cons cell, 20
consp, 43
constant
    symbolic, 363
create variable, 36
data abstraction, 355
data hiding, 356
data members, 356
data register, 445
data register direct
    addressing mode, 447
data segment, 225
defmacro, 32, 37, 41
defun, 32, 37, 41
defvar, 29, 40
delay, 96
delayed expression, 96
delete, 55
depth-first search, 135, 137
    vs breadth-first, 145
derived class, 360
desk checking, 313
destroy variable, 36
destructive, 12
destructive function, 13
dimension
    of array, 25
disjunctive normal form, 177
division, see / or truncate
DNF, 177
do, 48
do*, 49
doctor, 124
INDEX

dolist, 49
 dotimes, 50
dotted-pair, 20
   box notation, 21
   cell, 21
doubly recursive, 75
dribble, 53
dynamic binding, 29, 30
eager evaluation, 97
egrep, 424
eight-puzzle, 160
element
   of array, 25
ELIZA, 123
empty clause, 189
empty string, 422, 425
environment of a function, 238
eq, 43
eql, 43
equal, 43
error, 52
eval, 28, 29, 37, 46
evaluation of s-expressions, 34
evenp, 43
exponent, 212
expt, 45
factor, 394
fatal error, 444
fboundp, 37
fib-o.asm, 243
fib-o.asm (complete listing), 276
fib-s.asm, 240
fib-t.asm, 241
FIFO, 143
filtering, 122
final state, 425
find-ancestor, 116
first, 5
first-in-first-out, 143
flet, 32, 41
floating point, 212
floor, 45
force, 96
funcall, 32, 42
function, 40
function binding, 18
function call, 35
function calling conventions, 238
function definition
   global, 31
   local, 31
function overloading, 368
functional, 104
   programming language, 104
garbage, 12
garbage collector, 12
generate, 129
get, 47, 114
grammar, 126, 394, 395
grammar for regular expressions, 423
graph
   weighted, 162
halfword, 214
head
   of queue, 143
header file, 313
heapsort, 328
hello.asm, 225
hello.asm (complete listing), 226, 263
heuristic comparison function, 170
hexadecimal, 209

IEEE 754 floating point, 212
if, 48
immediate
   addressing mode, 447
imperative, 104
implies (⇒), 171
inconsistent formula, 173
indexing
   of array, 25
inheritance, 117
initial state, 425
instance, 356
instances, 382
instructions
   assembly language, 446
interpreter, 3
      LISP, 3
:initial-element, 38
iteration, 71
knowledge networks, 114
label, 457
labels, 220
labels, 32, 41
lambda, 40, 41
lambda abstraction, 69
lambda list, 34
larger.asm, 227
larger.asm (complete listing), 229
last-in-first-out, 139
lazy evaluation, 97
length, 46
let, 28, 29, 39
let*, 39
lexeme, 395
lexical, 395
lexical analyzer, 396, 438
lexical binding, 28, 30
LIFO, 139
list, 2
list, 46
list*, 46
listp, 43
literal, 176
load, 52
Load instructions, 254
load/store, 250
logical complement, 206
long, 214
lookahead, 408
looping, 230
macro, 32, 367
macro call, 33, 35
macro expansion, 33, 35
macro-expanded, 367
macroexpand, 33, 42
macrolet, 32, 42
main (program start), 220
make-array, 38
make-clause-set, 185
makunbound, 37
mantissa, 212
mapcar, 42
match, 122
matches
   string, 423
member, 43, 44
member functions, 356
memoizing, 90
message, 357
missing pointer, 324
mod, 45
move
   in state-space, 146
multiples.asm, 230
multiples.asm (complete listing), 264
multiplication, see *
namespace, 362
nconc, 55
NDFA, 425
network
ako, 114
non-deterministic finite automaton, 425
non-terminal, 394
non-terminal symbol, 127
not, 43
not (~), 171
nreverse, 55, 100
nth, 46
null, 43
number, 2, 394
object, 356
object oriented
   programming paradigm, 353, 377
object-oriented, 356
octal, 209
oddp, 43
one’s complement, 206
one-token lookahead, 408
operator overloading, 368
or, 48
or (∨), 171
overflow, 204
palindrome.asm, 231
palindrome.asm (complete listing), 266
parse, 395
parse tree, 406
parser, 396, 406
parsing, 406
pass-by-reference, 370
path, 135
pattern matching, 118
   example, 123
pop, 139, 446
positive closure, 423
pprint, 53
prin1, 51
princ, 51
print, 51
*print-circle*, 51
*print-pretty*, 51
printf.asm (complete listing), 272
procedural
   programming paradigm, 353, 354
   procedural abstractions, 66
   programming paradigm, 354
property, 113
   name, 113
   value, 113
propositional
   variable, 171
pseudoinstruction, 251, 258
Pure LISP, 100
purely functional, 89
push, 139, 446
putprop, 113
quadword, 214
queue, 143
quote, 45
random, 45
rank
   of array, 25
ratio
   produced by
      integer division, 45
RE, 423
read, 51
recoverable error, 444
recursion, 61
recursive rule, 61
recursive-descent parser, 408
reduce-to-and-or-not, 179
reference, 370
register, 445
regular expression, 422
reject, 425, 426
rem, 45
remove, 46
rest, 5
&rest, 34
RISC, 250
root, 115
of parse tree, 406
root class, 360
round, 45
rplaca, 55
rplacd, 55
runtime system, 390
s-expression, 1, 23
array, 24
evaluation of, 34
satisfiable formula, 173
sbrk syscall, 246
scanner, 396
search, 135
semantic, 395
semantics, 396, 414
set, 28, 37
setf, 13, 28, 29
setf
  aref, 38
car, 55
cdr, 55
got, 47
of global variable, 40
of local variable, 39
symbol-value, 37
sign bit, 212
signed binary numbers, 204
significand, 212
singly recursive, 75
sort, 55
special flag, 18
special form, 19, 33, 35
special symbol, 18, 27, 30, 40
sqrt, 45
square root, see sqrt
stack, 139
state, 146
state-space, 146
state-space search, 146
statement, 393
stream, 92
streams package, 372
string, 422
subformula, 194
subtraction, see –
symbol, 2, 3
  excluded characters, 2
  quotes and escapes, 3
symbol-function, 32, 37, 42
symbol-plist, 37
symbol-value, 37
symbolic atom, 393
symbolic constant, 363
symbolic expression, 1
syntax error, 444
syscall, 221
tail
  of queue, 143
tail-recursive, 72
INDEX

target type, 311
tautology, 173
term, 394
terminal, 394
terminal symbol, 127
terpri, 51
text segment, 225
theorem, 173
throw, 50
time, 53
token, 395, 396
top down design, 316
trace, 53
treesort.asm (complete listing), 278
truncate, 45
truth-assignments, 173
two’s complement, 206
typechecked, 353
union, 423
unit clause, 200
unreachable, 12
unsatisfiable formula, 173
unsigned binary numbers, 201
untrace, 53
unwind-protect, 50
utility functions, 404

variable
  binding, 18
  evaluation, 28, 39, 40
    LISP, 11
virtual, 390
virtual machine, 258

weighted graph, 162
word, 214
zerop, 43