Pl12sols - Solution manual

Principles of Programming Languages (COMSATS University Islamabad)

Instructor's Solutions Manual

to

Concepts of Programming Languages

Twelfth Edition

R.W. Sebesta

Preface

Changes for the Twelfth Edition

The goals, overall structure, and approach of this twelfth edition of *Concepts of Programming Languages* remain the same as those of the eleven earlier editions. The principal goals are to introduce the fundamental constructs of contemporary programming languages and to provide the reader with the tools necessary for the critical evaluation of existing and future programming languages. A secondary goal is to prepare the reader for the study of compiler design, by providing an in-depth discussion of programming language structures, presenting a formal method of describing syntax, and introducing approaches to lexical and syntactic analysis.

The twelfth edition evolved from the eleventh through several different kinds of changes. To maintain the currency of the material, nearly all of the discussion of some programming languages, specifically Lua and Objective-C, has been removed. Material on the newer language, Swift, was added to several chapters.

In addition, a new section on optional types was added to Chapter 6. Material was added to Section 8.3.4 to describe iterators in Python. In numerous places in the manuscript small changes were made to correct and/or clarify the discussion.

The Vision

This book describes the fundamental concepts of programming languages by discussing the design issues of the various language constructs, examining the design choices for these constructs in some of the most common languages, and critically comparing design alternatives.

Any serious study of programming languages requires an examination of some related topics, among which are formal methods of describing the syntax and semantics of programming languages, which are covered in Chapter 3. Also, implementation techniques for various language constructs must be considered: Lexical and syntax analysis are discussed in Chapter 4, and implementation of subprogram linkage is covered in Chapter 10. Implementation of some other language constructs is discussed in various other parts of the book.

The following paragraphs outline the contents of the twelfth edition.

Chapter Outlines

Chapter 1 begins with a rationale for studying programming languages. It then discusses the criteria used for evaluating programming languages and language constructs. The primary influences on language design, common design trade-offs, and the basic approaches to implementation are also examined.

Chapter 2 outlines the evolution of the languages that are discussed in this book. Although no attempt is made to describe any language completely, the origins, purposes, and contributions of each are discussed. This historical overview is valuable, because it provides the background necessary to understanding the practical and theoretical basis for contemporary language design. It also motivates further study of language design and evaluation. Because none of the remainder of the book depends on Chapter 2, it can be read on its own, independent of the other chapters.

Chapter 3 describes the primary formal method for describing the syntax of programming language—BNF. This is followed by a description of attribute grammars, which describe both the syntax and static semantics of languages. The difficult task of semantic description is then explored, including brief introductions to the three most common methods: operational, denotational, and axiomatic semantics.

Chapter 4 introduces lexical and syntax analysis. This chapter is targeted to those Computer Science departments that no longer require a compiler design course in their curricula. Similar to Chapter 2, this chapter stands alone and can be studied independently of the rest of the book, except for Chapter 3, on which it depends.

Chapters 5 through 14 describe in detail the design issues for the primary constructs of programming languages. In each case, the design choices for several example languages are presented and evaluated. Specifically, Chapter 5 covers the many characteristics of variables, Chapter 6 covers data types, and Chapter 7 explains expressions and assignment statements. Chapter 8 describes control statements, and Chapters 9 and 10 discuss subprograms and their implementation. Chapter 11 examines data abstraction facilities. Chapter 12 provides an in-depth discussion of language features that support object-oriented programming (inheritance and dynamic method binding), Chapter 13 discusses concurrent program units, and Chapter 14 is about exception handling, along with a brief discussion of event handling.

Chapters 15 and 16 describe two of the most important alternative programming paradigms: functional programming and logic programming. However, some of the data structures and control constructs of functional programming languages are discussed in Chapters 6 and 8. Chapter 15 presents an introduction to Scheme, including descriptions of some of its primitive functions, special forms, and functional forms, as well as some examples of simple functions written in Scheme. Brief introductions to ML, Haskell, and F# are given to illustrate some different directions in functional language design. Chapter 16 introduces logic programming and the logic programming language, Prolog.

To the Instructor

Chapters 1 and 3 are typically covered in detail, and though students find it interesting and beneficial reading, Chapter 2 receives little lecture time due to the lack of hard technical content. Because no material in subsequent chapters depends on Chapter 2, it can be skipped entirely. If a course in compiler design is required, Chapter 4 is not covered.

Chapters 5 through 9 should be relatively easy for students with extensive programming experience in C++, Java, or C#. Chapters 10 through 14 are more challenging and require more detailed lectures.

Chapters 15 and 16 are entirely new to most students at the junior level. Ideally, language processors for Scheme and Prolog should be available for students required to learn the material in these chapters. Sufficient material is included to allow students to dabble with some simple programs.

Undergraduate courses will probably not be able to cover all of the material in the last two chapters. Graduate courses, however, should be able to completely discuss the material in those chapters by skipping over some parts of the early chapters on imperative languages.

Supplemental Materials

The following supplements are available to all readers of this book at *www.pearsonhighered.com/cs-resources/*.

- **A set of lecture note slides. PowerPoint slides are available for each chapter in the book.**
- All of the figures from the book.

A companion Web site to the book is available at *www.pearson.com/cs-resources/*. This site contains mini-manuals (approximately 100-page tutorials) on a handful of languages.

Solutions to many of the problem sets are available to qualified instructors in our Instructor Resource Center at *www.pearson.com*. Please contact your school's Pearson Education representative or visit *www.pearson.com to register*.

Language Processor Availability

Processors for and information about some of the programming languages discussed in this book can be found at the following Web sites:

C, C++, Fortran, and Ada	*gcc.gnu.org*
C# and F#	*microsoft.com*
Java	*java.sun.com*
Haskell	*haskell.org*
Scheme	*www.plt-scheme.org/software/drscheme*
Perl	*www.perl.com*
Python	*www.python.org*
Ruby	*www.ruby-lang.org*

JavaScript is included in virtually all browsers; PHP is included in virtually all Web servers.

All this information is also included on the companion Web site.

Acknowledgments

The suggestions from outstanding reviewers contributed greatly to this book's present form and contents. In alphabetical order, they are:

Aaron Rababaah	*University of Maryland at Eastern Shore*
Amar Raheja	*California State Polytechnic University–Pomona*
Amer Diwan	*University of Colorado*
Bob Neufeld	*Wichita State University*
Bruce R. Maxim	*University of Michigan–Dearborn*
Charles Nicholas	*University of Maryland–Baltimore County*
Cristian Videira Lopes	*University of California–Irvine*
Curtis Meadow	*University of Maine*
David E. Goldschmidt	
Donald Kraft	*Louisiana State University*
Duane J. Jarc	*University of Maryland, University College*
Euripides Montagne	*University of Central Florida*
Frank J. Mitropoulos	*Nova Southeastern University*
Gloria Melara	*California State University–Northridge*
Hossein Saiedian	*University of Kansas*
I-ping Chu	*DePaul University*
Ian Barland	*Radford University*
K. N. King	*Georgia State University*
Karina Assiter	*Wentworth Institute of Technology*
Mark Llewellyn	*University of Central Florida*

Matthew Michael Burke	
Michael Prentice	*SUNY Buffalo*
Nancy Tinkham	*Rowan University*
Neelam Soundarajan	*Ohio State University*
Nigel Gwee	*Southern University–Baton Rouge*
Pamela Cutter	*Kalamazoo College*
Paul M. Jackowitz	*University of Scranton*
Paul Tymann	*Rochester Institute of Technology*
Richard M. Osborne	*University of Colorado–Denver*
Richard Min	*University of Texas at Dallas*
Robert McCloskey	*University of Scranton*
Ryan Stansifer	*Florida Institute of Technology*
Salih Yurttas	*Texas A&M University*
Saverio Perugini	*University of Dayton*
Serita Nelesen	*Calvin College*
Simon H. Lin	*California State University–Northridge*
Stephen Edwards	*Virginia Tech*
Stuart C. Shapiro	*SUNY Buffalo*
Sumanth Yenduri	*University of Southern Mississippi*
Teresa Cole	*Boise State University*
Thomas Turner	*University of Central Oklahoma*
Tim R. Norton	*University of Colorado–Colorado Springs*
Timothy Henry	*University of Rhode Island*
Walter Pharr	*College of Charleston*
Xiangyan Zeng	*Fort Valley State University*

Numerous other people provided input for the previous editions of *Concepts of Programming Languages* at various stages of its development. All of their comments were useful and greatly appreciated. In alphabetical order, they are:

Vicki Allan, Henry Bauer, Carter Bays, Manuel E. Bermudez, Peter Brouwer, Margaret Burnett, Paosheng Chang, Liang Cheng, John Crenshaw, Charles Dana, Barbara Ann Griem, Mary Lou Haag, John V. Harrison, Eileen Head, Ralph C. Hilzer, Eric Joanis, Leon Jololian, Hikyoo Koh, Jiang B. Liu, Meiliu Lu, Jon Mauney, Robert McCoard, Dennis L. Mumaugh, Michael G. Murphy, Andrew Oldroyd, Young Park, Rebecca Parsons, Steve J. Phelps, Jeffery Popyack, Steven Rapkin, Hamilton Richard, Tom Sager, Raghvinder Sangwan, Joseph Schell, Sibylle Schupp, Mary Louise Soffa, Neelam Soundarajan, Ryan Stansifer, Steve Stevenson, Virginia Teller, Yang Wang, John M. Weiss, Franck Xia, and Salih Yurnas.

Matt Goldstein, Portfolio Management Specialist; Meghan Jacoby, Portfolio Management Assistant; Managing Content Producer, Scott Disanno; and Prathiba Rajagopal, all deserve my gratitude for their efforts to produce the twelfth edition both quickly and carefully.

About the Author

Robert Sebesta is an Associate Professor Emeritus in the Computer Science Department at the University of Colorado–Colorado Springs. Professor Sebesta received a BS in applied mathematics from the University of Colorado in Boulder and MS and PhD degrees in computer science from Pennsylvania State University. He taught computer science for more than 40 years..

Contents

Answers to Selected Problems

Chapter 1

Problem Set:

3. Some arguments for having a single language for all programming domains are: It would dramatically cut the costs of programming training and compiler purchase and maintenance; it would simplify programmer recruiting and justify the development of numerous language dependent software development aids.

4. Some arguments against having a single language for all programming domains are: The language would necessarily be huge and complex; compilers would be expensive and costly to maintain; the language would probably not be very good for any programming domain, either in compiler efficiency or in the efficiency of the code it generated. More importantly, it would not be easy to use, because regardless of the application area, the language would include many unnecessary and confusing features and constructs (those meant for other application areas). Different users would learn different subsets, making maintenance difficult.

5. One possibility is wordiness. In some languages, a great deal of text is required for even simple complete programs. For example, COBOL is a very wordy language. In Ada, programs require a lot of duplication of declarations. Wordiness is usually considered a disadvantage, because it slows program creation, takes more file space for the source programs, and can cause programs to be more difficult to read.

7. The argument for using the right brace to close all compounds is simplicity—a right brace always terminates a compound. The argument against it is that when you see a right brace in a program, the location of its matching left brace is not always obvious, in part because all multiple-statement control constructs end with a right brace.

8. The reasons why a language would distinguish between uppercase and lowercase in its identifiers are: (1) So that variable identifiers may look different than identifiers that are names for constants, such as the convention of using uppercase for constant names and using lowercase for variable names in C, and (2) so that catenated words as names can have their first letter distinguished, as in `TotalWords`. (Some think it is better to include a connector, such as underscore.) The primary reason why a language would not

distinguish between uppercase and lowercase in identifiers is it makes programs less readable, because words that look very similar are actually completely different, such as `SUM` and `Sum`.

10. One of the main arguments is that regardless of the cost of hardware, it is not free. Why write a program that executes slower than is necessary. Furthermore, the difference between a well-written efficient program and one that is poorly written can be a factor of two or three. In many other fields of endeavor, the difference between a good job and a poor job may be 10 or 20 percent. In programming, the difference is much greater.

15. The use of type declaration statements for simple scalar variables may have very little effect on the readability of programs. If a language has no type declarations at all, it may be an aid to readability, because regardless of where a variable is seen in the program text, its type can be determined without looking elsewhere. Unfortunately, most languages that allow implicitly declared variables also include explicit declarations. In a program in such a language, the declaration of a variable must be found before the reader can determine the type of that variable when it is used in the program.

18. The main disadvantage of using paired delimiters for comments is that it results in diminished reliability. It is easy to inadvertently leave off the final delimiter, which extends the comment to the end of the next comment, effectively removing code from the program. The advantage of paired delimiters is that you *can* comment out areas of a program. The disadvantage of using only beginning delimiters is that they must be repeated on every line of a block of comments. This can be tedious and therefore error-prone. The advantage is that you cannot make the mistake of forgetting the closing delimiter.

Chapter 2

Problem Set:

6. Because of the simple syntax of LISP, few syntax errors occur in LISP programs. Unmatched parentheses is the most common mistake.

7. The main reason why imperative features were put in LISP was to increase its execution efficiency.

10. The main motivation for the development of PL/I was to provide a single tool for computer centers that must support both scientific and commercial applications. IBM believed that the needs of the two classes of applications were merging, at least to some degree. They felt that the simplest solution for a provider of systems, both hardware and software, was to furnish a single hardware system running a single programming language that served both scientific and commercial applications.

11. IBM was, for the most part, incorrect in its view of the future of the uses of computers, at least as far as languages are concerned. Commercial applications are nearly all done in languages that are specifically designed for them. Likewise for scientific applications. On the other hand, the IBM design of the 360 line of computers was a great success--it still dominates the area of computers between supercomputers and minicomputers. Furthermore, 360 series computers and their descendants have been widely used for both scientific and commercial applications. These applications have been done, in large part, in Fortran and COBOL.

14. The argument for typeless languages is their great flexibility for the programmer. Literally any storage location can be used to store any type value. This is useful for very low-level languages used for systems programming. The drawback is that type checking is impossible, so that it is entirely the programmer's responsibility to insure that expressions and assignments are correct.

18. A good deal of restraint must be used in revising programming languages. The greatest danger is that the revision process will continually add new features, so that the language grows more and more complex. Compounding the problem is the reluctance, because of existing software, to remove obsolete features.

22. One situation in which pure interpretation is acceptable for scripting languages is when the amount of computation is small, for which the processing time will be negligible. Another situation is when the amount of computation is relatively small and it is done in an interactive environment, where the processor is often idle because of the slow speed of human interactions.

23. New scripting languages may appear more frequently than new compiled languages because they are often smaller and simpler and focused on more narrow applications, which means their libraries need not be nearly as large.

Chapter 3

Instructor's Note:

In the program proof on page 160, there is a statement that may not be clear to all, specifically, `(n + 1)* … * n = 1`. The justification of this statement is as follows:

Consider the following expression:

```
(count + 1) * (count + 2) * … * n
```

The former expression states that when `count` is equal to `n`, the value of the later expression is `1`. Multiply the later expression by the quotient:

```
(1 * 2 * … * count) / (1 * 2 * … * count)
```

whose value is `1`, to get

```
(1 * 2 * … * count * (count + 1) * (count + 2) * … * n) /

      (1 * 2 * … * count)
```

The numerator of this expressions is `n!`. The denominator is `count!`. If `count` is equal to `n`, the value of the quotient is

```
n! / n!
```

or `1`, which is what we were trying to show.

Problem Set:

2a. <class_head> → {<modifier>} **class** <id> [**extends** class_name]

[**implements** <interface_name> {, <interface_name>}]

<modifier> → **public** | **abstract** | **final**

2c. <switch_stmt> → **switch** (<expr>) {**case** <literal> **:** <stmt_list>

{**case** <literal> : <stmt_list> } [**default :** <stmt_list>] }

3. <assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> * <term>

| <term>

<term> → <factor> + <term>

| <factor>

<factor> → (<expr>)

| <id>

6.

(a) <assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = A * <expr>

=> A = A * (<expr>)

=> A = A * (<id> + <expr>)

=> A = A * (B + <expr>)

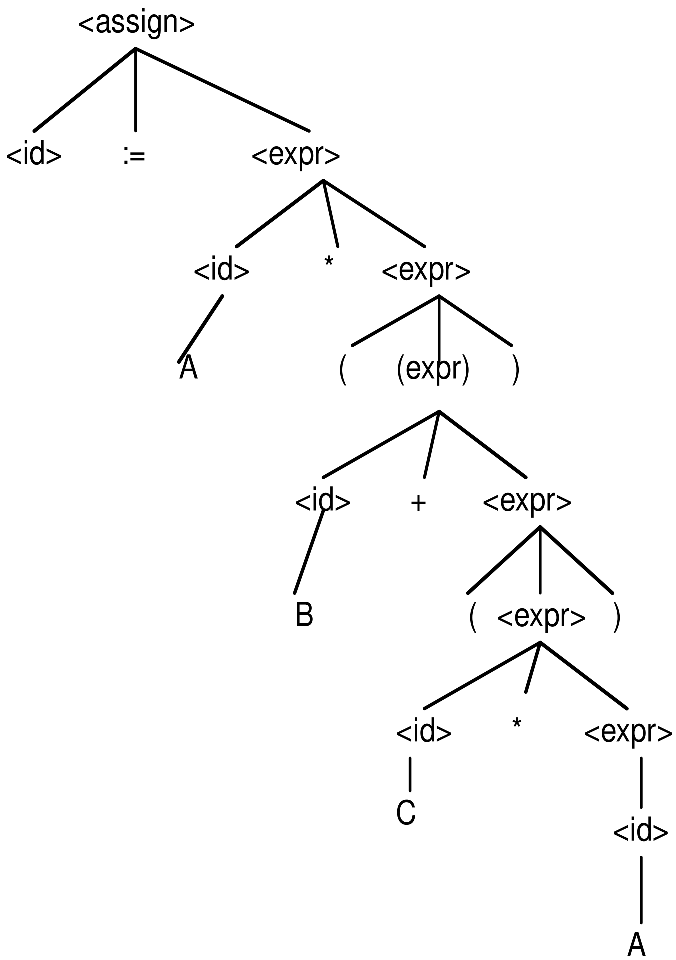
=> A = A * (B + (<expr>))

=> A = A * (B + (<id> * <expr>))

=> A = A * (B + (C * <expr>))

=> A = A * (B + (C * <id>))

=> A = A * (B + (C * A))



7.

(a) <assign> => <id> = <expr>

=> A = <expr>

=> A = <term>

=> A = <factor> * <term>

=> A = (<expr>) * <term>

=> A = (<expr> + <term>) * <term>

=> A = (<term> + <term>) * <term>

=> A = (<factor> + <term>) * <term>

=> A = (<id> + <term>) * <term>

=> A = (A + <term>) * <term>

=> A = (A + <factor>) * <term>

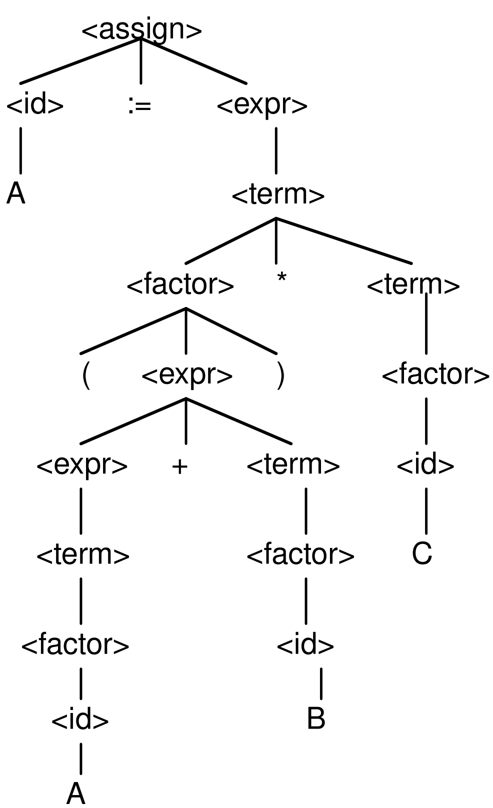
=> A = (A + <id>) * <term>

=> A = (A + B) * <term>

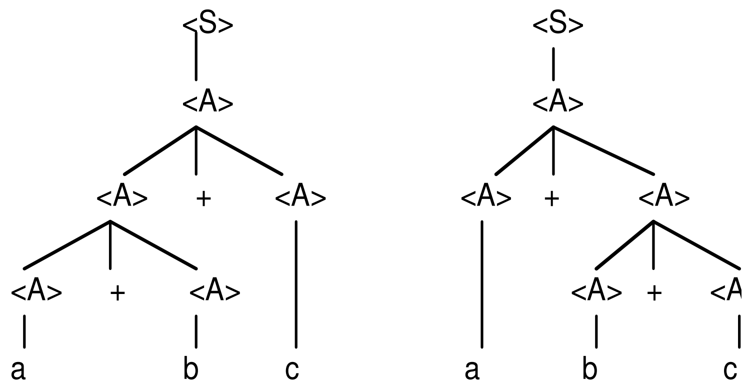
=> A = (A + B) * <factor>

=> A = (A + B) * <id>

=> A = (A + B) * C



8. The following two distinct parse tree for the same string prove that the grammar is ambiguous.



9. Assume that the unary operators can precede any operand. Replace the rule

<factor> → <id>

with

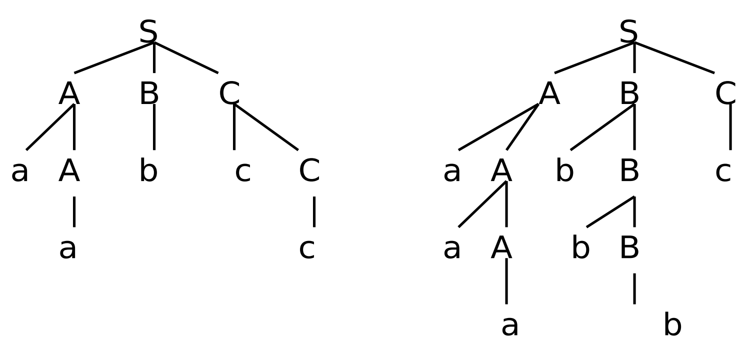
<factor> → + <id>

| - <id>

10. One or more a's followed by one or more b's followed by one or more c's.

13. S → a S b | a b

14.



16. <assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> (+ | -) <expr>

| (<expr>)

| <id>

18. The value of an intrinsic attribute is supplied from outside the attribute evaluation process, usually from the lexical analyzer. A value of a synthesized attribute is computed by an attribute evaluation function.

19. Replace the second semantic rule with:

<var>[2].env ← <expr>.env

<var>[3].env ← <expr>.env

<expr>.actual_type ← <var>[2].actual_type

predicate: <var>[2].actual_type = <var>[3].actual_type

21.

(a) (Java **do-while**) We assume that the logic expression is a single relational expression.

loop: (do body)

if <relational_expression> goto out

goto loop

out: ...

(b) (Ada **for**) **for** I **in** first .. last **loop**

I = first

loop: if I < last goto out

...

I = I + 1

```
            goto loop
    out:    ...
```

(c) (Fortran Do)

```
            K = start
    loop:   if K > end goto out
            ...
            K = K + step
            goto loop
    out:    ...
```

(e) (C **for**) **for** (expr1; expr2; expr3) ...

```
            evaluate(expr1)
    loop:   control = evaluate(expr2)
            if control == 0 goto out
            ...
            evaluate(expr3)
            goto loop
    out:    ...
```

22a. M_{pf}(for var in init_expr .. final_expr loop L end loop, s) $\triangleq$

if VARMAP(i, s) = **undef** for var or some i in init_expr or final_expr

then **error**

else if M_e(init_expr, s) > M_e(final_expr, s)

then s

else M_l(while init_expr - 1 <= final_expr do L, M_a(var := init_expr + 1, s))

22b. M_r(repeat L until B) $\triangleq$

if M_b(B, s) = **undef**

then **error**

else if M_{sl}(L, s) = **error**

then **error**

else if M_b(B, s) = **true**

then M_{sl}(L, s)

else M_r(repeat L until B), M_{sl}(L, s))

22c. M_b(B, s) $\triangleq$ if VARMAP(i, s) = **undef** for some i in B

then **error**

else B', where B' is the result of

evaluating B after setting each

variable i in B to VARMAP(i, s)

22d. M_{cf}(for (expr1; expr2; expr3) L, s) $\triangleq$

if VARMAP (i, s) = **undef** for some i in expr1, expr2, expr3, or L

then **error**

else if M_e (expr2, M_e (expr1, s)) = 0

then s

else M_{help} (expr2, expr3, L, s)

M_{help} (expr2, expr3, L, s) $\triangleq$

if VARMAP (i, s) = **undef** for some i in expr2, expr3, or L

then **error**

else

if M_{sl} (L, s) = **error**

then s

else M_{help} (expr2, expr3, L, M_{sl} (L, M_e (expr3, s))

23.

(a) a = 2 * (b - 1) - 1 {a > 0}

2 * (b - 1) - 1 > 0

2 * b - 2 - 1 > 0

2 * b > 3

b > 3 / 2

(b) b = (c + 10) / 3 {b > 6}

(c + 10) / 3 > 6

c + 10 > 18

c > 8

(c) a = a + 2 * b - 1 {a > 1}

a + 2 * b - 1 > 1

2 * b > 2 - a

b > 1 - a / 2

(d) x = 2 * y + x - 1 {x > 11}

2 * y + x - 1 > 11

2 * y + x > 12

24.

(a) a = 2 * b + 1

b = a - 3 {b < 0}

$a - 3 < 0$

$a < 3$

Now, we have:

$a = 2 * b + 1 \ \{a < 3\}$

$2 * b + 1 < 3$

$2 * b + 1 < 3$

$2 * b < 2$

$b < 1$

(b) $a = 3 * (2 * b + a);$

$b = 2 * a - 1 \ \{b > 5\}$

$2 * a - 1 > 5$

$2 * a > 6$

$a > 3$

Now we have:

$a = 3 * (2 * b + a) \ \{a > 3\}$

$3 * (2 * b + a) > 3$

$6 * b + 3 * a > 3$

$2 * b + a > 1$

$n > (1 - a) / 2$

Chapter 4

Problem Set:

1.

(a) FIRST(aB) = {a}, FIRST(b) = {b}, FIRST(cBB) = {c}, Passes the test

(b) FIRST(aB) = {a}, FIRST(bA) = {b}, FIRST(aBb) = {a}, Fails the test

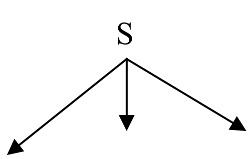
(c) FIRST(aaA) = {a}, FIRST(b) = {b}, FIRST(caB) = {c}, Passes the test

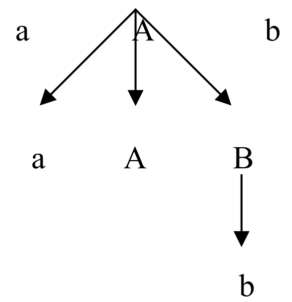
3. a + b * c

```
Call lex  /* returns a */
Enter <expr>
Enter <term>
Enter <factor>
Call lex  /* returns + */
Exit <factor>
Exit <term>
Call lex  /* returns b */
Enter <term>
Enter <factor>
Call lex  /* returns * */
Exit <factor>
Call lex  /* returns c */
Enter <factor>
Call lex  /* returns end-of-input */
Exit <factor>
Exit <term>
Exit <expr>
```

5.

(a) aaAbb



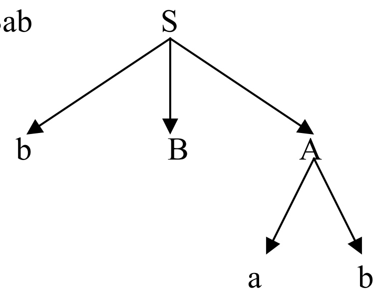


Phrases: aaAbb, aAb, b

Simple phrases: b

Handle: b

(b) bBab



Phrases: bBab, ab

Simple phrases: ab

Handle: ab

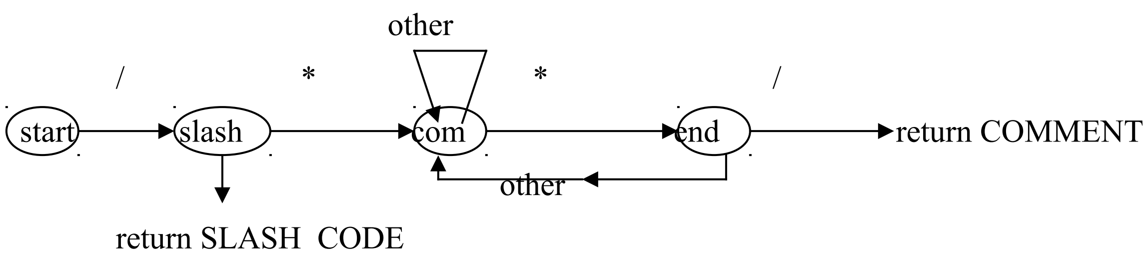
7.

Stack	*Input*	*Action*
0	id * (id + id) $	Shift 5
0id5	* (id + id) $	Reduce 6 (Use GOTO[0, F])
0F3	* (id + id) $	Reduce 4 (Use GOTO[0, T])
0T2	* (id + id) $	Reduce 2 (Use GOTO[0, E])
0T2*7	(id + id) $	Shift 7
0T2*7(4	id + id) $	Shift 4
0T2*7(4id5	+ id) $	Shift 5
0T2*7(4F3	+ id) $	Reduce 6 (Use GOTO[4, F])
0T2*7(4T2	+ id) $	Reduce 4 (Use GOTO[4, T])

0T2*7(4E8	+ id) $	Reduce 2 (Use GOTO[4, E])
0T2*7(4E8+6	id) $	Shift 6
0T2*7(4E8+6id5	) $	Shift 5
0T2*7(4E8+6F3	) $	Reduce 6 (Use GOTO[6, F])
0T2*7(4E8+6T9	) $	Reduce 4 (Use GOTO[6, T])
0T2*7(4E8	) $	Reduce 1 (Use GOTO[4, E])
0T2*7(4E8)11	$	Shift 11
0T2*7F10	$	Reduce 5 (Use GOTO[7, F])
0T2	$	Reduce 5 (Use GOTO[0, T])
0E1	$	Reduce 2 (Use GOTO[0, E])
--ACCEPT--		

Programming Exercises:

1. Every arc in this graph is assumed to have `addChar` attached. Assume we get here only if `charClass` is `SLASH`.



3.
```
int getComment() {

  getChar();

/* The slash state */

  if (charClass != AST)

    return SLASH_CODE;

  else {

/* The com state-end state loop */
```

```
    do {
      getChar();
/* The com state loop */
      while (charClass != AST)
        getChar();
    } while (charClass != SLASH);
    return COMMENT;
}
```

Chapter 5

Problem Set:

2. The advantage of a typeless language is flexibility; any variable can be used for any type values. The disadvantage is poor reliability due to the ease with which type errors can be made, coupled with the impossibility of type checking detecting them.

4. Implicit heap-dynamic variables acquire types only when assigned values, which must be at runtime. Therefore, these variables are always dynamically bound to types.

5. Suppose that a Fortran subroutine is used to implement a data structure as an abstraction. In this situation, it is essential that the structure persist between calls to the managing subroutine.

6.

(a) i. `sub1`

ii. `sub1`

iii. `main`

(b) i. `sub1`

ii. `sub1`

iii. `sub1`

7. Static scoping: `x is 5`

Dynamic scoping: `x is 10`

8. Variable Where Declared

In `sub1`:

a	sub1
y	sub1
z	sub1
x	main

In `sub2`:

a	sub2
b	sub2
z	sub2
y	sub1
x	main

In `sub3`:

a	sub3
x	sub3
w	sub3
y	main
z	main

10. Point 1:

a	1
b	2
c	2
d	2

Point 2:

a	1
b	2
c	3
d	3
e	3

Point 3: same as Point 1

Point 4:

a	1
b	1
c	1

11. Variable Where Declared

(a)	`d, e, f`	`fun3`
	`c`	`fun2`
	`b`	`fun1`
	`a`	`main`
(b)	`d, e, f`	`fun3`
	`b, c`	`fun1`
	`a`	`main`
(c)	`b, c, d`	`fun1`
	`e, f`	`fun3`
	`a`	`main`
(d)	`b, c, d`	`fun1`
	`e, f`	`fun3`
	`a`	`main`
(e)	`c, d, e`	`fun2`
	`f`	`fun3`
	`b`	`fun1`
	`a`	`main`
(f)	`b, c, d`	`fun1`
	`e`	`fun2`
	`f`	`fun3`
	`a`	`main`

12.	Variable	Where Declared
(a)	`a, x, w`	`sub3`
	`b, z`	`sub2`
	`y`	`sub1`
(b)	`a, x, w`	`sub3`
	`y, z`	`sub1`
(c)	`a, y, z`	`sub1`
	`x, w`	`sub3`
	`b`	`sub2`
(d)	`a, y, z`	`sub1`

```
      x, w        sub3
```

(e)
```
      a, b, z     sub2
      x, w        sub3
      y           sub1
```

(f)
```
      a, y, z     sub1
      b           sub2
      x, w        sub3
```

Chapter 6

Problem Set:

1. Boolean variables stored as single bits are very space efficient, but on most computers access to them is slower than if they were stored as bytes.

2. Integer values stored in decimal waste storage in binary memory computers, simply as a result of the fact that it takes four binary bits to store a single decimal digit, but those four bits are capable of storing 16 different values. Therefore, the ability to store six out of every 16 possible values is wasted. Numeric values can be stored efficiently on binary memory computers only in number bases that are multiples of 2. If humans had developed hands with a number of fingers that was a power of 2, these kinds of problems would not occur.

5. When implicit dereferencing of pointers occurs only in certain contexts, it makes the language slightly less orthogonal. The context of the reference to the pointer determines its meaning. This detracts from the readability of the language and makes it slightly more difficult to learn.

7. The only justification for the `->` operator in C and C++ is writability. It is slightly easier to write `p -> q` than `(*p).q`.

9. Let the subscript ranges of the three dimensions be named `min(1)`, `min(2)`, `min(3)`, `max(1)`, `max(2)`, and `max(3)`. Let the sizes of the subscript ranges be `size(1)`, `size(2)`, and `size(3)`. Assume the element size is 1.

Row Major Order:

location(`a[i,j,k]`) = (address of `a[min(1),min(2),min(3)]`)

```
+((i-min(1))*size(3) + (j-min(2)))*size(2) + (k-min(3))
```

Column Major Order:

location(`a[i,j,k]`) = (address of `a[min(1),min(2),min(3)]`)

```
+((k-min(3))*size(1) + (j-min(2)))*size(2) + (i-min(1))
```

10. The advantage of this scheme is that accesses that are done in order of the rows can be made very fast; once the pointer to a row is gotten, all of the elements of the row can

be fetched very quickly. If, however, the elements of a matrix must be accessed in column order, these accesses will be much slower; every access requires the fetch of a row pointer and an address computation from there. Note that this access technique was devised to allow multidimensional array rows to be segments in a virtual storage management technique. Using this method, multidimensional arrays could be stored and manipulated that are much larger than the physical memory of the computer.

14. Implicit heap storage recovery eliminates the creation of dangling pointers through explicit deallocation operations, such as **`delete`**. The disadvantage of implicit heap storage recovery is the execution time cost of doing the recovery, often when it is not even necessary (there is no shortage of heap storage).

20. Static type checking is better than dynamic type checking for two reasons: First, anything done at compile time leads to better overall efficiency, simply because production programs are often executed but far less often compiled. Second, type checking uncovers program errors, and the earlier errors are found the less costly it is to remove them.

21. A language that allows many type coercions can weaken the beneficial effect of strong typing by allowing many potential type errors to be masked by simply coercing the type of an operand from its incorrect type given in the statement to an acceptable type, rather than reporting it as an error.

Chapter 7

Problem Set:

1. Suppose `Type1` is a subrange of `Integer`. It may be useful for the difference between `Type1` and `Integer` to be ignored by the compiler in an expression.

7. An expression such as `a + fun(b)`, as described on page 300.

8. Consider the integer expression `A + B + C`. Suppose the values of `A`, `B`, and `C` are 20,000, 25,000, and -20,000, respectively. Further suppose that the machine has a maximum integer value of 32,767. If the first addition is computed first, it will result in overflow. If the second addition is done first, the whole expression can be correctly computed.

9.

(a) $(((a * b)^{1} - 1)^{2} + c)^{3}$

(b) $(((a * (b - 1)^{1})^{2} / c)^{3} \text{ mod } d)^{4}$

(c) $(((a - b)^{1} / c)^{2} \ \& \ (((d * e)^{3} / a)^{4} - 3)^{5})^{6}$

(d) $(((- a)^{1} \text{ or } (c = d)^{2})^{3} \text{ and } e)^{4}$

(e) $((a > b)^{1} \text{ xor } (c \text{ or } (d <= 17)^{2})^{3})^{4}$

(f) $(- (a + b)^{1})^{2}$

10.

(a) (a * (b - (1 + c)^{1})^{2})^{3}

(b) (a * ((b - 1)^{2} / (c mod d)^{1})^{3})^{4}

(c) ((a - b)^{5} / (c & (d * (e / (a - 3)^{1})^{2})^{3})^{4})^{6}

(d) (- (a or (c = (d and e)^{1})^{2})^{3})^{4}

(e) (a > (xor (c or (d <= 17)^{1})^{2})^{3})^{4}

(f) (- (a + b)^{1})^{2}

11. <expr> → <expr> or <e1> | <expr> xor <e1> | <e1>

<e1> → <e1> and <e2> | <e2>

<e2> → <e2> = <e3> | <e2> /= <e3> | <e2> < <e3>

| <e2> <= <e3> | <e2> > <e3> | <e2> >= <e3> | <e3>

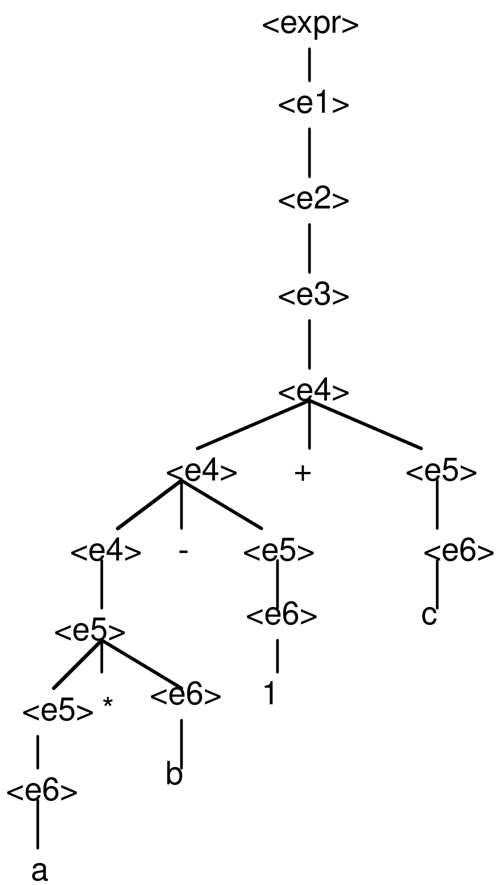
<e3> → <e4>

<e4> → <e4> + <e5> | <e4> - <e5> | <e4> & <e5> | <e4> mod <e5> | <e5>

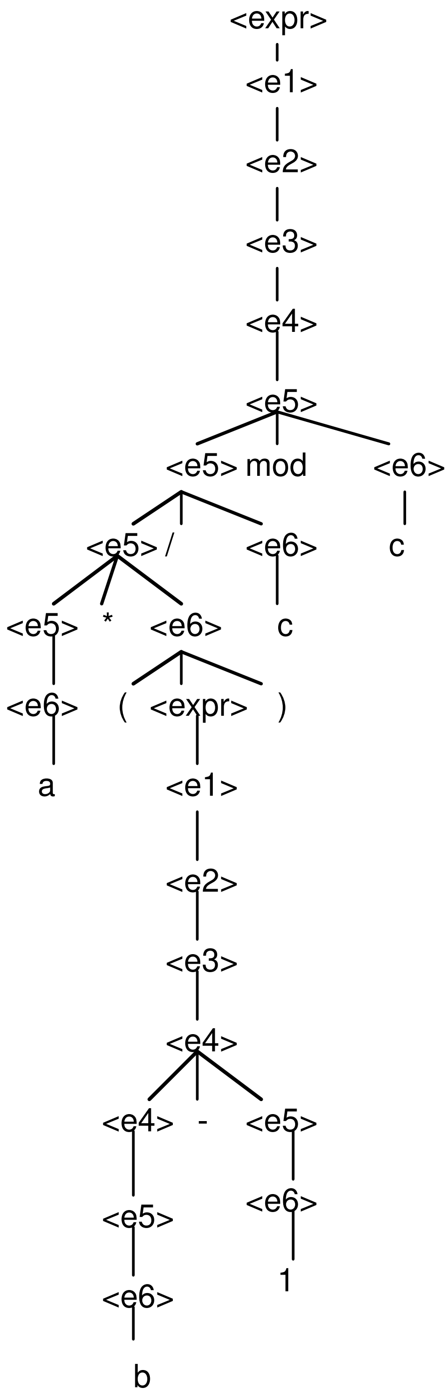
<e5> → <e5> * <e6> | <e5> / <e6> | not <e5> | <e6>

<e6> → a | b | c | d | e | const | (<expr>)

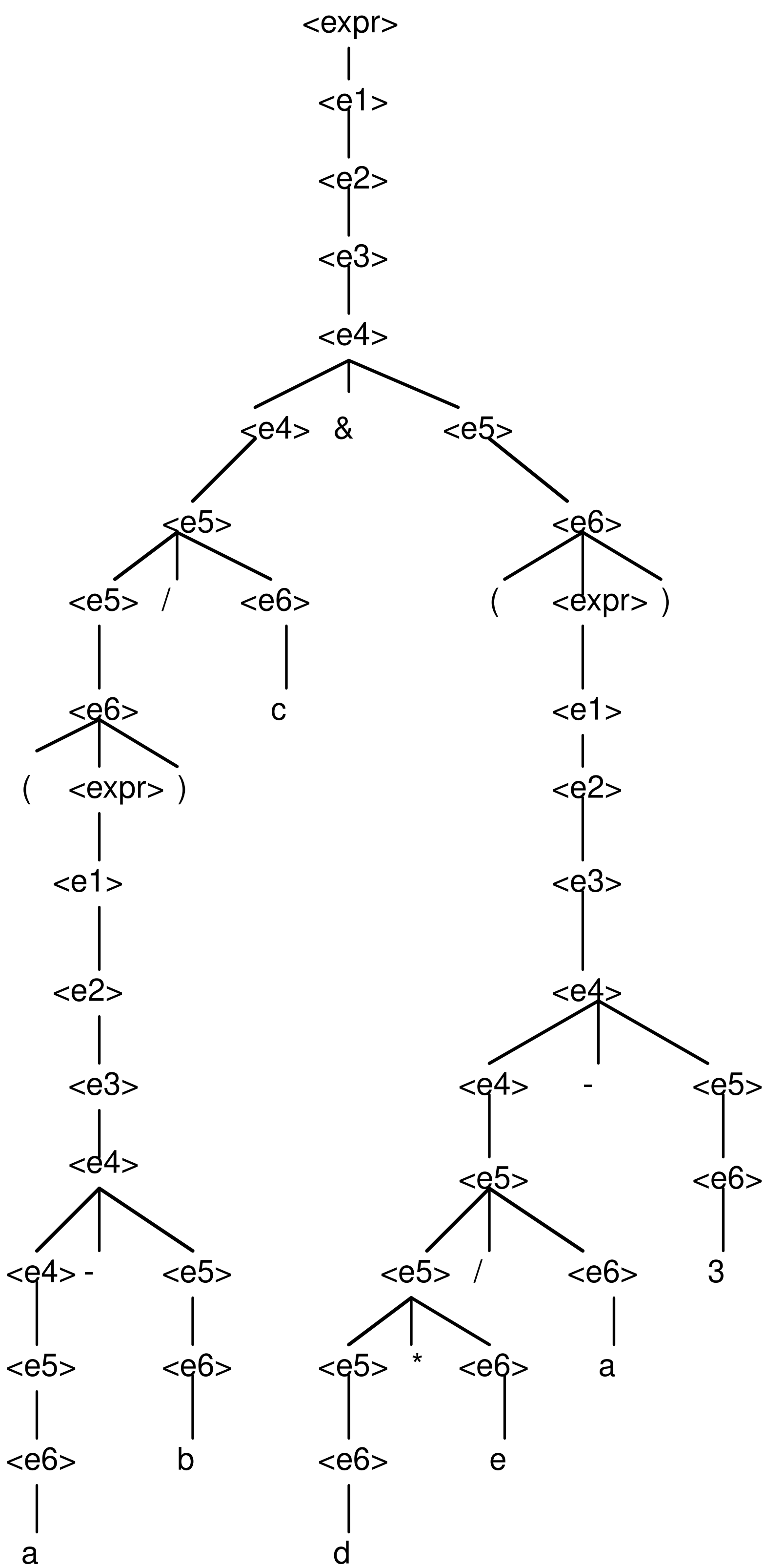
12. (a)



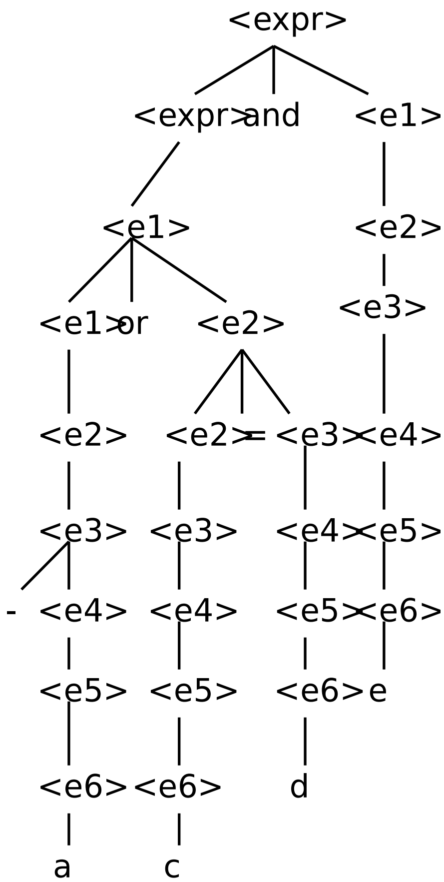
12. (b)


<expr>
<e1>
<e2>
<e3>
<e4>
<e5>
<e5> mod <e6>
<e5> / <e6> c
<e5> * <e6> c
<e6> (<expr>)
a <e1>
<e2>
<e3>
<e4>
<e4> - <e5>
<e5> <e6>
<e6> 1
b

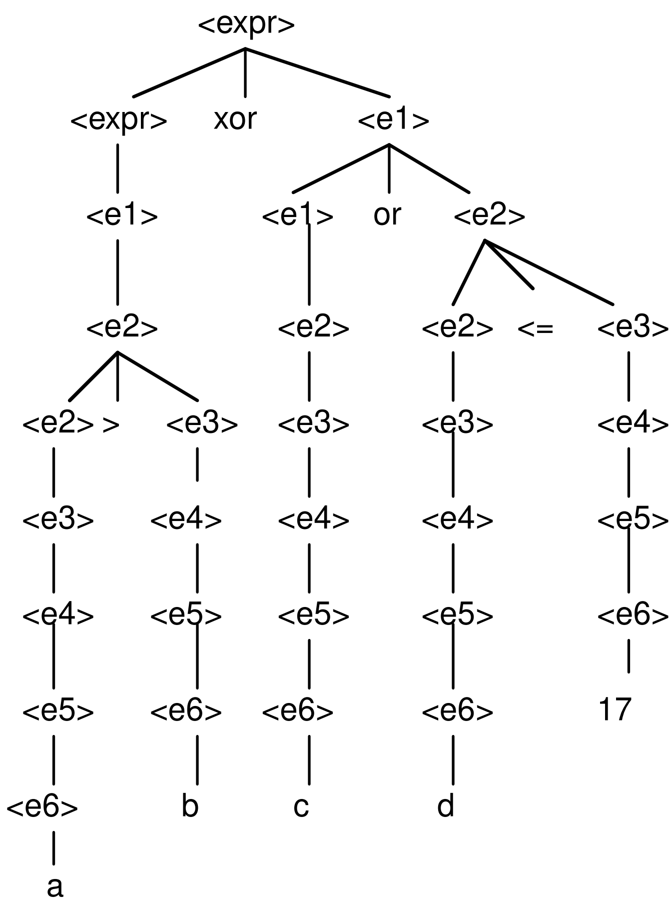
12. (c)


<expr>
<e1>
<e2>
<e3>
<e4>
<e4> & <e5>
<e5> <e6>
<e5> / <e6> (<expr>)
<e6> c <e1>
(<expr>) <e2>
<e1> <e3>
<e2> <e4>
<e3> <e4> - <e5>
<e4> <e5> <e6>
<e4> - <e5> <e5> / <e6> 3
<e5> <e6> <e5> * <e6> a
<e6> b <e6> e
a d

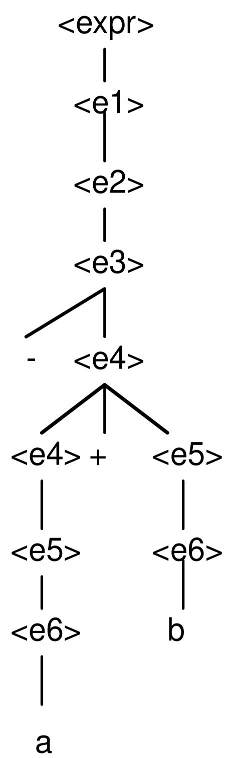
12. (d)


<expr>
<expr> and <e1>
<e1>
<e2>
<e1> or <e2>
<e3>
<e2> <e2> = <e3> <e4>
<e3> <e3> <e4> <e5>
- <e4> <e4> <e5> <e6>
<e5> <e5> <e6> e
<e6> <e6> d
a c

12. (e)


<expr>
<expr> xor <e1>
<e1> <e1> or <e2>
<e2> <e2> <e2> <= <e3>
<e2> > <e3> <e3> <e3> <e4>
<e3> <e4> <e4> <e4> <e5>
<e4> <e5> <e5> <e5> <e6>
<e5> <e6> <e6> <e6> 17
<e6> b c d
a

12. (f)



13. (a) (left -> right) `sum1` is `46`; `sum2` is `48`

(b) (right -> left) `sum1` is `48`; `sum2` is `46`

15. One reason functional side effects would be difficult to remove from C is that all of C's subprograms are functions, providing the ability of returning only a single data value (though it could be an array). The problem is that in many cases it is necessary (or at least convenient) to return more than one data value, which is done through the use of pointer actual parameters, which are a means of creating functional side effects.

20. (a) 7

(b) 12

21. Java specifies that operands in expressions are all evaluated in left-to-right order to eliminate the problem described in Section 7.2.2.1.

Chapter 8

Problem Set:

1. Three situations in which a combined counting and logical control loops are:

a. A list of values is to be added to a SUM, but the loop is to be exited if SUM exceeds some prescribed value.

b. A list of values is to be read into an array, where the reading is to terminate when either a prescribed number of values have been read or some special value is found in the list.

c. The values stored in a linked list are to be moved to an array, where values are to be moved until the end of the linked list is found or the array is filled, whichever comes first.

4. Unique closing keywords on compound statements have the advantage of readability and the disadvantage of complicating the language by increasing the number of keywords.

5. One argument in favor of Python's use of indentation is that it demands that programmers use an indentation scheme that promotes readability. This results is a rigid standard for program layout that ensures that all Python programs will be equally readable. If indentation is good for readability, why not use it to indicate compound statements? It is difficult to find a strong argument against the use of indentation to indicate program structure, although sloppy programmers will find the required discipline annoying.

9. The primary argument for using Boolean expressions exclusively as control expressions is the reliability that results from disallowing a wide range of types for this use. In C, for example, an expression of any type can appear as a control expression, so typing errors that result in references to variables of incorrect types are not detected by the compiler as errors.

12. There are two possible reasons why control can be transferred into a C loop construct: First, there may have been some particular situation in the first large application of C, which was UNIX, where it was convenient. Recall that C was originally designed specifically for writing UNIX. Second, little was known about security in programming constructs (and consequently little concern about it) at the time C was designed, so it is unlikely that it was thought to be a problem.

Programming Exercises:

1.

(a)

```
Do K = (J + 13) / 27, 10
  I = 3 * (K + 1) - 1
End Do
```

(b)

```
for k in (j + 13) / 27 .. 10 loop
 i := 3 * (k + 1) - 1;
end loop;
```

(c)
```
for (k = (j + 13) / 27; k <= 10; i = 3 * (++k) - 1)
```

2.

(a)
```
Do K = (J + 13.0) / 27.0, 10.0, 1.2
  I = 3.0 * (K + 1.2) - 1.0
End Do
```

(b)
```
while (k <= 10.0) loop
  i := 3.0 * (k + 1.2) - 1.0;
  k := k + 1.2;
end loop;
```

(c)
```
for (k = (j + 13.0) / 27.0; k <= 10.0;
   k = k + 1.2, i = 3.0 * k - 1)
```

3.

(a)
```
Select Case (k)
  Case (1, 2)
    J = 2 * K - 1
  Case (3, 5)
    J = 3 * K + 1
  Case (4)
    J = 4 * K - 1
  Case (6, 7, 8)
    J = K - 2
  Case Default
    Print *, 'Error in Select, K = ', K
End Select
```

(b)
```
case k is
  when 1 | 2 => j := 2 * k - 1;
```

```
    when 3 | 5 => j := 3 * k + 1;
    when 4 => j := 4 * k - 1;
    when 6..8 => j := k - 2;
    when others =>
      Put ("Error in case, k =');
      Put (k);
      New_Line;
  end case;
```

(c)
```
switch (k)
  {
   case 1: case 2:
     j = 2 * k - 1;
     break;
   case 3: case 5:
     j = 3 * k + 1;
     break;
   case 4:
     j = 4 * k - 1;
     break;
   case 6: case 7: case 8:
     j = k - 2;
     break;
   default:
     printf("Error in switch, k =%d\n", k);
   }
```

4.
```
j = -3;
key = j + 2;
```

```
for (i = 0; i < 10; i++){
   if ((key == 3) || (key == 2))
     j--;
   else if (key == 0)
     j += 2;
   else j = 0;
   if (j > 0)
     break;
   else j = 3 - i;
  }
```

5. (C)

```
for (i = 1; i <= n; i++) {
  flag = 1;
  for (j = 1; j <= n; j++)
    if (x[i][j] <> 0) {
      flag = 0;
      break;
     }
  if (flag == 1) {
    printf("First all-zero row is: %d\n", i);
    break;
   }
 }
```

(Ada)

```
for I in 1..N loop
  Flag := true;
```

```
    for J in 1..N loop
      if X(I, J) /= 0 then
        Flag := false;
        exit;
      end if;
    end loop;
    if Flag = true then
      Put("First all-zero row is: ");
      Put(I);
      Skip_Line;
      exit;
    end if;
  end loop;
```

7.

```
I, J : Integer;
N : Integer := 100;

I = 0;
J = 17;
while I < N loop
  Sum := Sum + I * J + 3;
  I : I + 1;
  J := J - 1;
end loop;
```

10.

```
if (x > 10) y = x;
else if (x < 5) y = 2 * x;
else if (x == 7) y = x + 10;
```

Chapter 9

Problem Set:

2. The main advantage of this method is the fast accesses to formal parameters in subprograms. The disadvantages are that recursion is rarely useful when values cannot be passed, and also that a number of problems, such as aliasing, occur with the method.

4. This can be done in both Java and C#, using a static (or class) data member for the page number.

5. Assume the calls are not accumulative; that is, they are always called with the initialized values of the variables, so their effects are not accumulative.

a.	b.	c.
2, 1, 3, 5, 7, 9	1, 2, 3, 5, 7, 9	1, 2, 3, 5, 7, 9
2, 1, 3, 5, 7, 9	2, 3, 1, 5, 7, 9	2, 3, 1, 5, 7, 9
2, 1, 3, 5, 7, 9	5, 1, 3, 2, 7, 9	5, 1, 3, 2, 7, 9 (unless the addresses of the actual parameters are recomputed on return, in which case there will be an index range error.)

6. It is rather weak, but one could argue that having both adds complexity to the language without sufficient increase in writability.

7. (a) 1, 3

   (b) 2, 6

   (c) 2, 6

11. Only its designers can answer this question definitively. The advantage of including an out mode for parameter passing is clear: If a parameter is used only to return a value from a subprogram, it is sensible to restrict its use to that. Such a parameter should not be allowed to have an initial value and it must be assigned a value before the subprogram terminates. These restrictions can only be enforced implicitly if a separate mode is included for such parameters. Given the other insecurities of C++, it is not surprising that it does not include an out mode. Java may not include out mode parameters because, at least initially, it was meant to be a simple language.

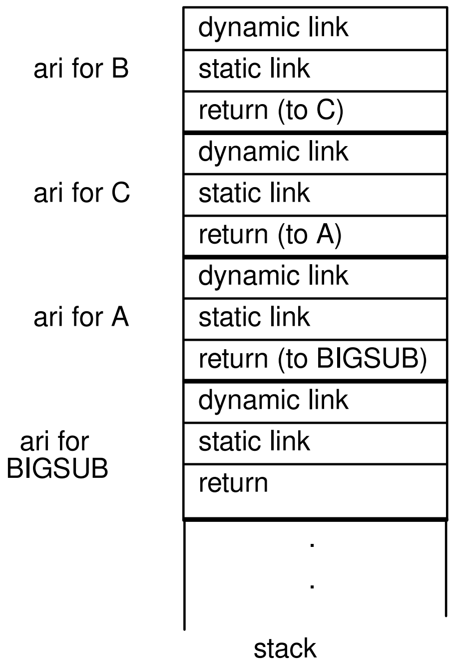
14. Many contemporary languages do not allow nested subprograms because many designers now believe that there are better ways to organize programs. Also, they think the additional complexity of nested subprograms outweighs their value. Finally, there is the problem of the nested structure of programs deteriorating through continued maintenance, leading to largely unstructured programs in the end, regardless of their initial structure.

15. Two arguments against pass-by-name parameters are: First, programs that use pass-by-name parameters can be overly complex and difficult to understand. Second, pass-by-name parameters are far less efficient than other parameter-passing methods.

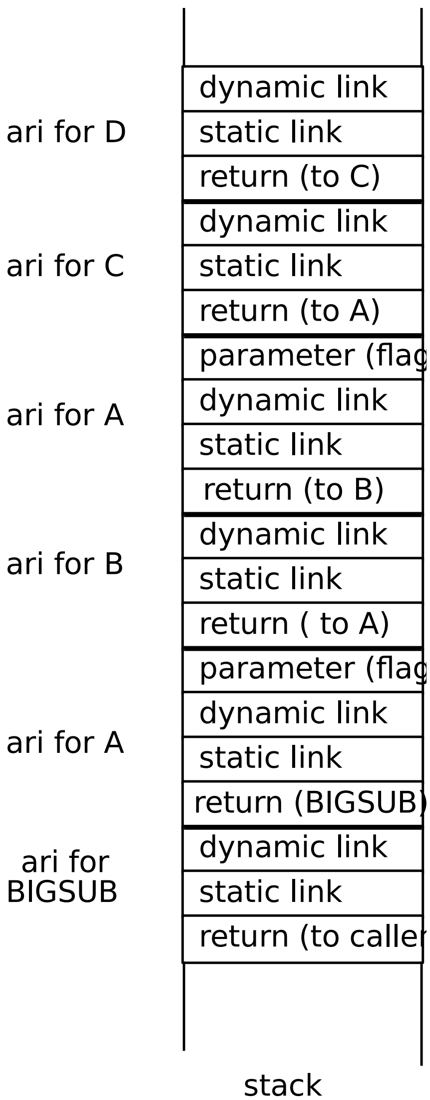
Chapter 10

Problem Set:

1.



3.


dynamic link
ari for D
static link
return (to C)
dynamic link
ari for C
static link
return (to A)
parameter (flag
dynamic link
ari for A
static link
return (to B)
dynamic link
ari for B
static link
return (to A)
parameter (flag
dynamic link
ari for A
static link
return (BIGSUB)
dynamic link
ari for BIGSUB
static link
return (to caller
stack

7. One very simple alternative is to assign integer values to all variable names used in the program. Then the integer values could be used in the activation records, and the comparisons would be between integer values, which are much faster than string comparisons.

8. Following the hint stated with the question, the target of every goto in a program could be represented as an address and a nesting_depth, where the nesting_depth is the difference between the nesting level of the procedure that contains the goto and that of the procedure containing the target. Then, when a goto is executed, the static chain is followed by the number of links indicated in the nesting_depth of the goto target. The stack top pointer is reset to the top of the activation record at the end of the chain.

9. Including two static links would reduce the access time to nonlocals that are defined in scopes two steps away to be equal to that for nonlocals that are one step away. Overall, because most nonlocal references are relatively close, this could significantly increase the execution efficiency of many programs.

11. There are two options for implementing blocks as parameterless subprograms: One way is to use the same activation record as a subprogram that has no parameters. This is the most simple way, because accesses to block variables will be exactly like accesses to local variables. Of course, the space for the static and dynamic links and the return address will be wasted. The alternative is to leave out the static and dynamic links and the return address, which saves space but makes accesses to block variables different from subprogram locals.

Chapter 11

Problem Set:

2. The problem with this is that the user is given access to the stack through the returned value of the "top" function. For example, if `p` is a pointer to objects of the type stored in the stack, we could have:

```
p = top(stack1);

*p = 42;
```

These statements access the stack directly, which violates the principle of a data abstraction.

5. There are several dangers inherent in C’s approach to encapsulation. First, the user is allowed to simply paste the contents of the header file into the application file, which can lead to using subsequently updated implementation files without using the potentially updated header file. This can cause type conflicts between the implementation file and the header file. Another problem with pasting the header file into the implementation file is the loss of documentation of the dependence of the implementation file on the header file.

6. C++ did not eliminate the problems described in Problem 8 because it uses the C linker on its programs.

8. Inlining a function or method makes them more efficient because it eliminates the cost of linkage. Allowing developers to specify methods and functions that may be inlined by the compiler is good because the developer would know better than anyone which methods are computationally small and frequently called, which would make them good candidates for inlining. One argument against the C++ policy on inlining is that the developer is really only suggesting that a function or method be inlined. The compiler may or may not inline such a subprogram, depending on whether it can determine whether it can be done. Another problem with the C++ policy on inlining is that it requires that non-inlined methods be defined outside the class, which eliminates the physical encapsulation of all of the class members.

11. The three ways a C++ client program can reference a name from a namespace defined in a header file are (assuming the namespace name is `MyStack` and the variable is named `topPtr`):

a. `MyStack::topPtr`

b. Including the statement:

```
using MyStack::topPtr;
```

in the program.

c. Including the statement:

```
using namespace MyStack;
```

in the program and referencing `topPtr` by its name.

13. The obvious advantage of being able to change objects in Ruby is programming flexibility. It allows a wide variety of different possibilities of object reuse. The disadvantage is complexity and readability. It is in a sense like dynamic typing—you cannot determine what an object really is statically.

Chapter 12

Problem Set:

1. SIMULA 67 classes did not implement information hiding, so it could not completely support abstract data types, which is an essential component of support for object-oriented programming.

3. In C++, a method can only be dynamically bound if all of its ancestors are marked `virtual`. Be default, all method binding is static. In Java, method binding is dynamic by default. Static binding only occurs if the method is marked `final`, which means it cannot be overriden.

5. C++ has extensive access controls to its class entities. Individual entities can be marked `public`, `private`, or `protected`, and the derivation process itself can impose further access controls by being `private`. Ada, on the other hand, has no way to restrict inheritance of entities (other than through child libraries, which this book does not describe), and no access controls for the derivation process.

8. Two problems of abstract data types that are ameliorated by inheritance are: (a) Reuse of abstract data types is much easier, because the modifications of existing types need not be done on the legacy code, and (b) with abstract data types, all types are independent and at the same level, disallowing any logically hierarchical type dependencies.

10. One disadvantage of inheritance is that types cannot be defined to be independent.

12. If a subclass has an is-a relationship with its parent class, a variable of the subclass can appear anywhere a variable of the parent class is allowed to appear.

16. One reason why all Java objects have a common ancestor is so they can all inherit a few universally useful methods.

17. The **`finalize`** clause in Java allows a specific action to take place, regardless of whether a **`try`** clause succeeds or fails.

20. A significant problem with multiple inheritance is that two of the parents can define a method with the same name and the same protocol. A class that implements an interface must define all of the methods declared in the interface. So, if both the parent class and the interface include methods with the same name and protocol, the subclass must reimplement that method, thereby avoiding the name conflict.

23. In Java, instance variables may be marked as having private, public, or protected access, meaning they are visible only in the class where defined, everywhere, or only in the class where defined and all of its subclasses, respectively. In Ruby, all instance variables are private by default, and that cannot be changed by the programmer.

Chapter 13

Problem Set:

1. Competition synchronization is not necessary when no actual concurrency takes place simply because there can be no concurrent contention for shared resources. Two nonconcurrent processes cannot arrive at a resource at the same time.

2. When deadlock occurs, assuming that only two program units are causing the deadlock, one of the involved program units should be gracefully terminated, thereby allowed the other to continue.

3. The main problem with busy waiting is that machine cycles are wasted in the process.

4. Deadlock would occur if the `release(access)` were replaced by a `wait(access)` in the consumer process, because instead of relinquishing access control, the consumer would wait for control that it already had.

6. Sequence 1:

`A` fetches the value of `BUF_SIZE` (6)
`A` adds 2 to the value (8)
`A` puts 8 in `BUF_SIZE`
`B` fetches the value of `BUF_SIZE` (8)
`B` subtracts 1 (7)
`B` put 7 in `BUF_SIZE`
`BUF_SIZE` = 7

Sequence 2:

`A` fetches the value of `BUF_SIZE` (6)
`B` fetches the value of `BUF_SIZE` (6)
`A` adds 2 (8)
`B` subtracts 1 (5)
`A` puts 8 in `BUF_SIZE`
`B` puts 5 in `BUF_SIZE`
`BUF_SIZE` = 5

Sequence 3:

`A` fetches the value of `BUF_SIZE` (6)
`B` fetches the value of `BUF_SIZE` (6)
`A` adds 2 (8)
`B` subtracts 1 (5)
`B` puts 5 in `BUF_SIZE`
`A` puts 8 in `BUF_SIZE`
`BUF_SIZE` = 8

Many other sequences are possible, but all produce the values 5, 7, or 8.

10. The safety of cooperation shynchronization using semaphores is basically the same as using Ada's **`when`** clauses, although the **`when`** clauses are somewhat more readable than semaphores.

Chapter 14

Problem Set:

1. The designers of C got efficiency for giving up subscript range checking.

2. Three approaches to exception handling in languages that do not provide direct support for it are: One is to send an auxiliary parameter that is used for the subprogram to indicate whether or not there was an error in the subprogram. Another is to pass an error label to the subprogram and have the subprogram return to that label, rather than to the statement immediately following the call. Still another is to send the name of an error handling subprogram as a parameter to the subprogram. If the subprogram detects an error, it calls that subprogram.

5. There are several advantages of a linguistic mechanism for handling exceptions, such as that found in Ada, over simply using a flag error parameter in all subprograms. One advantage is that the code to test the flag after every call is eliminated. Such testing makes programs longer and harder to read. Another advantage is that exceptions can be propagated farther than one level of control in a uniform and implicit way. Finally, there is the advantage that all programs use a uniform method for dealing with unusual circumstances, leading to enhanced readability.

6. There are several disadvantages of sending error handling subprograms to other subprograms. One is that it may be necessary to send several error handlers to some subprograms, greatly complicating both the writing and execution of calls. Another is that there is no method of propagating exceptions, meaning that they must all be handled locally. This complicates exception handling, because it requires more attention to handling in more places.

9. `throw i` in `fun1` is caught in `fun2`; `throw f` in `fun1` is caught in `fun1`; `throw j` in `fun2` is caught in the inner try block of `fun2`; `throw g` in `fun2` is caught in the outer try block of `fun2`.

12. The resumption model is useful when the exception is only an unusual condition, rather than an error. The termination model is useful when the exception is an error and it is highly unlikely that the error can be corrected so that execution could continue in some useful way.

Chapter 15

Problem Set :

8. Scheme cannot be a pure functional language if it includes `DISPLAY`, because `DISPLAY` has the side effect of producing output.

9. `y` returns the given list with leading elements removed up to but not including the first occurrence of the first given parameter.

10. `x` returns the number of non-`#f` atoms in the given list.

Programming Exercises:

5.
```
(DEFINE (deleteall atm lst)
 (COND
   ((NULL? lst) '())
   ((EQ? atm (CAR lst)) (deleteall atm (CDR lst)))
   (ELSE (CONS (CAR lst) (deleteall atm (CDR lst)))
 ))
```

7.
```
(DEFINE (deleteall atm lst)
 (COND
   ((NULL? lst) '())
   ((NOT (LIST? (CAR lst)))
      (COND
        ((EQ? atm (CAR lst)) (deleteall atm (CDR lst)))
        (ELSE (CONS (CAR lst) (deleteall atm (CDR lst))))
       ))
   (ELSE (CONS (deleteall atm (CAR lst))
                       (deleteall atm (CDR lst))))
 ))
```

11.
```
(DEFINE (reverse lis)
  (COND
    ((NULL? lis) '())
    (ELSE (APPEND (reverse (CDR lis)) (CONS (CAR lis) '() )))
))
```

13.
```
(DEFINE (union set1 set2)
```

```
    (COND ((NULL? set1) set2)
          ((MEMBER (CAR set1) set2) (union (CDR set1) set2))
          (ELSE (union (CDR set1) (CONS (CAR set1) set2))
                                 (CONS (CAR set1) set2))
  ))
```

21. Programming Exercise 11 in F#:

```
let rec reverse lis =
   if lis = [] then []
   else List.append (reverse (List.tail(lis))) (List.head(lis) :: [])
```

Chapter 16

Problem Set:

1. Ada variables are statically bound to types. Prolog variables are bound to types only when they are bound to values. These bindings take place during execution and are temporary.

2. On a single processor machine, the resolution process takes place on the rule base, one rule at a time, starting with the first rule, and progressing toward the last until a match is found. Because the process on each rule is independent of the process on the other rules, separate processors could concurrently operate on separate rules. When any of the processors finds a match, all resolution processing could terminate.

6. The list processing capabilities of Scheme and Prolog are similar in that they both treat lists as consisting of two parts, head and tail, and in that they use recursion to traverse and process lists.

7. The list processing capabilities of Scheme and Prolog are different in that Scheme relies on the primitive functions `CAR`, `CDR`, and `CONS` to build and dismantle lists, whereas with Prolog these functions are not necessary.

10. A well-formed formula of the predicate calculus is in *prenex normal form* if all of the quantifiers stand at the front and any other logical constant stands within the scope of all of the quantifiers.

A well-formed formula of the predicate calculus is in *Skolem normal form* if it is in prenex normal form and all of the existential quantifiers come first.

Programming Exercises:

2.
```
intersect([], X, []).

intersect([X | R], Y, [X | Z] :-

   member(X, Y),

   !,

   intersect(R, Y, Z).

intersect([X | R], Y, Z) :- intersect(R, Y, Z).
```

Note: this code assumes that the two lists, `X` and `Y`, contain no duplicate elements.

3.
```
union([], X, X).

union([X | R], Y, Z) :- member(X, Y), !, union(R, Y, Z).

union([X | R], Y, [X | Z]) :- union(R, Y, Z).
```