Student Edition

Instructor's Guide to PROGRAMMING LANGUAGES Design and Implementation Third Edition

Marvin V. Zelkowitz
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD

PRENTICE HALL, Upper Saddle River, New Jersey 07458

This Postscript file contains chapter 5 of the *Instructor's Guide* to *Programming Languages: Design and Implementation (Third Edition)*, Prentice Hall, Upper Saddle River, NJ (1996). Chapter 5 contains new material on text processing languages including Postscript, HTML, and Java. For a complete copy of the *Instructor's Guide*, containing solutions to problems for each chapter, additional problems with solutions, and sample syllabuses, contact your Prentice Hall representative.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN X-XX-XXXXXX-X

Prentice-Hall International (UK) Limited, London
Prentice-Hall of Australia Pty. Limited, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall Hispanoamericana, S. A., Mexico
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Simon & Schuster Asia Pte. Ltd., Singapore
Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

Contents

Chapter 5

Word Processing Applications

Note to reader. This represents some initial ideas of where this author thinks language design is evolving over the next several years. This chapter represents an early draft of a possible new chapter for a fourth edition of *Programming Languages: Design and Implementation*, should a fourth edition be produced in the future. You may wish to include this material as an additional topic near the end of the course. A Postscript version of this chapter is available on the WWW at:

http://www.cs.umd.edu/users/mvz/pzbook/

Background and related material in the current text related to these topics include the following sections:

- Section 4.2.5 on international character sets for discussion of Postscript.
- Section 6.2 on postfix, and page 248 on the language Forth for the discussion on Postscript.
- Section 9.6.2 on client server computing for the discussion on the World Wide Web and HTML.
- Section 9.6.3 on desktop publishing for the discussion on Postscript and the World Wide Web and HTML.
- Sections 12.2 (C++) and 12.3 (Smalltalk) for the discussion of Java.

5.1 Introduction

Computers are often thought of as electronic *calculators* or *number*-crunchers due to their initial applications of computing ballistics tables, processing payrolls, or solving differential equations in order to predict the weather. However, the nature of computing is changing as we use computers today to fly airplanes, drive cars, play games, process documents, and speak to us. In keeping with these trends to non-numeric applications, language design is following suit. Some of this has been discussed in the existing text. Section 13.1 on LISP and Chapter 14 on Prolog both represent languages useful for inferring facts about data. In this chapter, we look at the evolution of textual information. Although Sections 9.6 presents some initial concepts on networking, client/server computing, and desktop publishing, the explosion of the Internet and the growth of the World Wide Web has greatly increased interest in languages for processing text. In this chapter we will look at Postscript as a language for describing documents, and we will then look at HTML (HyperText Markup Language) and Java as languages for passing information around the Internet.

Translator models. Translators can be classified by the output they produce in three ways:

1. **Instance.** The output is a specification of a solution to the problem posed to the translator. For example, given the question "What is the meaning of life, time, and everything?"[1] the translator will simply write the number 42.[2] This corresponds to an interpreter that executes the program to produce the answer.

2. **Dynamic description.** In this case, the translator produces an algorithm (i.e., a program) that will compute the answer. This corresponds to the typical compiler. A C compiler will produce a program for a hardware-implemented machine that can execute the program to produce the desired answer. We have several common machine architectures (e.g., Intel Pentium, Apple PowerPC, various RISC processors). A compiler will develop the set of instructions unique to that specific machine architecture that will be used to compute the answer.

3. **Static description.** The translator produces a description of the output. For example, the output from the question "What is the meaning of life, time, and everything?" could be $\langle$integer$\rangle$ $\langle$binary value$\rangle$ 101010 $\langle$end$\rangle$. Note that this is more general than the first output given earlier. The first output would be 42, but this may not be a unique answer. For example, on the planet Frammis, where inhabitants count in quinary (base 5), the digits 42 represent the binary number 10110, which is our decimal number 22. That is a very different value. However, the static description could be interpreted using any base, and printed using any character set, not necessarily the usual Arabic numerals we are used to.

For numeric calculations, most of the world has adapted to the Arabic numerals and Roman character set. However, the same is not true for word-processing applications. Readers of Japanese, Hebrew, Arabic, Korean, and the numerous other character sets need to have text presented in their own alphabet.

For textual applications, then, the three forms of translators given above have the following interpretation:

[1] From Douglas Adams' brilliant book *Hitchhiker's Guide to the Galaxy* [ADAMS 1979].

[2] Don't ask me; ask Mr. Adams.

1. **Instance.** This would represent the specific output format. For text it could be a set of bits representing the page to be displayed. For a page that is 1000 bits wide by 2000 bits long, 2,000,000 bits would be required for a black and white description of this page, or about 250K bytes of data. Although data compression techniques can reduce this amount significantly, it still represents an inflexible method of creating and storing such information. In general, this approach is not used for these applications.

2. **Dynamic description.** In this model, the translator converts the document into an executable program that executes to create its own image. This is the process used by Postscript. Postscript is an executable language that has the effect of "painting" a blank sheet of paper. When the paper is finished, a print command causes the page to be displayed. Translators that produce Postscript produce a program that creates the final document in its formatted description. This is then executed by a Postscript interpreter (either as part of a terminal display or part of a printer) in order to view each page. In Section 5.2 of this *IG* we describe Postscript further.

3. **Static description.** In this case we describe what attributes the final document has, but not the details of how it actually looks. For example, Microsoft uses RTF (Rich Text Format) for saving documents using its Word word-processing applications. Other processors can be written to read and write RTF format. This provides a series of commands describing each attribute of the document, much like the description of the number 42 as an integer and binary number given earlier. For world wide web use, HTML has become the standard description of text. In Section 5.3 of this *IG* we describe HTML, and in Section 5.3.3 of this *IG* we look at Java as a potential extension to the document processing capabilities of web access.

5.2 Postscript

Postscript was developed by John Warnock and Chuck Geschke of Adobe Systems [ADOBE 1985] in the early 1980s. It was initially used as the "printing engine" of Apple computers, but soon became a widely used standard for most computer systems. Postscript interpreters (either as software or as hardware components) exist for printing documents on almost every current computer system in use today.

Postscript uses the model of painting text (or pictures) onto a blank page. When the page is finished, it is printed and a new page is started. This is the dynamic description method of translation given earlier. Each Postscript document consists of a program that prints successive pages onto a printer (or display screen).

A Postscript program consists of four components:

1. An interpreter for performing calculations. A simple postfix execution stack is the basic model.

Language Summary 6.2: Forth

Features: Postfix source language leads to an efficient execution model, even though generally interpreted. System runs on two stacks — a subroutine return stack and an expression evaluation stack. Run-time model very small making it useful on small embedded computers.

History: Forth was developed by Charles Moore around 1970. The name was a contraction of "Fourth Generation Programming Language" with the program name limited to five characters. The language was a replacement for FORTRAN on small minicomputers in the 1970s where space was at a premium and the only input/output device was often a very slow and cumbersome paper tape. Having a resident translator/interpreter made for easy program development on the target system.

Example: Program to compute: $1^2 + 2^2 + \ldots + 9^2 + 10^2$

(Notation: a, b, c is expression stack. c is stack(top))
: SQR DUP * ; (Defines $square$ by: $n \Rightarrow n,\ n \Rightarrow (n * n)$)
: DOSUM SWAP 1 + SWAP OVER SQR + ; ($N,\ S \Rightarrow N+1,\ S+(N+1)^2$)
($N, S \Rightarrow S, N \Rightarrow S, (N+1) \Rightarrow (N+1), S \Rightarrow$
$(N+1), S, (N+1) \Rightarrow (N+1), S, (N+1)^2 \Rightarrow$
$(N+1),\ S+(N+1)^2$)
3 6 DOSUM . . 22 4 ok (Period (.) prints stack(top). Output is $22 = 4^2 + 6$)
0 0 10 0 DO DOSUM LOOP . 385 ok (Apply DOSUM from 0 to 9 (Stop at 10))

Reference: E. Rather, D. Colburn, and C. Moore, "The evolution of Forth," *ACM History of Programming Languages Conference II*, Cambridge, MA (April 1993) *(SIGPLAN Notices (28)3 [March 1993])*, 177–199.

Figure 5.1. Forth Language (from page 248).

2. A language syntax. This is based upon Forth (Figure 5.1 of this *IG*).
3. An extension to Forth with "painting" commands for managing the process of painting text and pictures on a sheet of paper.
4. A series of conventions, not part of the formal Postscript language, that various printers use for consistency in presentation. Use of these conventions makes it easier for transporting Postscript documents from one system to another.

Each component will be described in the sections that follow.

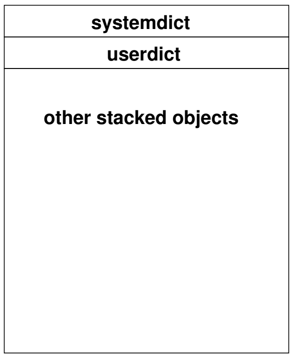

Figure 5.2. Postscript execution model.

5.2.1 Postscript interpreter

A Postscript program consists a sequence of commands that represent the postfix of a program. This sequence manipulates a stack, as shown in Figure 5.2 of this *IG*.

The Postscript stack begins with two entries initially on the stack, which the program may not remove:

- *Systemdict* is the system dictionary, which represents the initial binding of Postscript objects to their internal representation.
- *Userdict* is the user dictionary, which represents the new definitions included within this execution of a Postscript program. This may include redefinition of primitive objects already defined in *systemdict*.

Execution stacks. Postscript actually manages four stacks:

1. The *operand* stack contains the operands as they are stacked, executed, and unstacked.
2. The *dictionary* stack contains only dictionary objects. This stack defines the scope and context of each definition.
3. The *execution* stack contains executable objects. For the most part these are functions in intermediate stages of execution.
4. The *graphics state* stack manages the context for painting objects on the page and will be discussed in Section 5.2.3 of this *IG*.

Note: Comments in Postscript begin with % and continue until the end of line.

5.2.2 The Postscript language

A Postscript program is written as a sequence of ASCII characters. The program is broken down into a sequence of tokens. As each token is read, its definition is accessed in the stack (by first looking in *userdict* and then *systemdict*) and then executed by an appropriate action.

Primitive data. Postscript data consists of the following objects:

Numbers can be integers (e.g., 1, 3, -42) or reals (e.g., -1.234, 1.2E-10).

Names are executable objects and generally consist of any characters except whitespace (e.g., AName, abc, A.Name). In general, names cause their values to be looked up in the dictionary stack and their values executed. Thus the name *add* causes the definition in *systemdict* to be accessed. This definition causes the top two operand stack entries are added together. Of course the user can write a new definition for *add* and place it in *userdict* in order to redefine the *add* primitive.

Literals are names preceded with a slash (/). This causes the name to be placed on the operand stack rather than having its value looked up. This allows the l-value of *add* (i.e., its name) to be placed in the stack rather than its r-value (i.e., the add procedure) invoked. For example, when defining a new function, before the definition is entered in the dictionary it is first necessary to place the name and its definition in the operand stack and then execute the name *def*. The name *def*, when looked up in *systemdict* will cause the top two entries in the operand stack (which should be the new function name and its executable definition) to be added as a new entry in *userdict*.

Compound data. Compound data consists of various components grouped together. These consist of:

Arrays are primitive objects in brackets ([,]). For example:

```
[123 4 567] % An array of 3 integers
    (Comments begin with %)
[123 abc /xyz] % An array of an integer, a name, and a literal name
```

Strings are any sequence of characters enclosed in parentheses:

```
(This is a string)
```

Procedures are sequences of Postscript tokens enclosed in set braces ({, }):

```
{dup mul} % Squares a number by duplicating the top of stack and
        then multiplying by itself.
```

As an example of a Postscript program, Figure 5.3 of this *IG* is the Postscript version of the Forth program in Figure 5.1 of this *IG*.

Line 1. A comment starts with %.

Line 2. The name (literal) Helvetica is placed on the operand stack and then *findfont* replaces the name with the font definition on the operand stack. Helvetica is a fairly standard sans-serif font. Times-Roman is the usual serif font.

Line 3. *scalefont* scales the Helvetica font to 20 points. Normal text is usually

```
 1: %Same as Forth program on page 248 of text
 2: /Helvetica findfont
 3: 20 scalefont
 4: setfont
 5: 200 400 moveto
 6: /formatit {10 10 string cvrs show} def
 7: /sqr {dup mul} def
 8: /dosum {exch 1 add exch 1 index sqr add} def
 9: 3 6 dosum 2 copy formatit ( ) show formatit
10: clear
11: 200 375 moveto
12: 0 0 0 1 9 {pop dosum} for formatit
13: showpage
```

Figure 5.3. Postscript program.

11 or 12 point. Distances are measured in points, 72 to the inch.

Line 4. The current font on the operand stack (Helvetica scaled to 20 points) is made the current font. All text painted on the page will use this font until changed.

Line 5. The cursor is moved to coordinates (200,400), or roughly 3 inches from the left margin and almost 6 inches from the bottom of the page. The origin (location (0,0)) is assumed to be the lower left corner of the page.

Line 6. *formatit* is defined as a function in *userdict* that places the integer at the top of the operand stack onto the page. It creates a string of 10 zeros (10 *string*), takes the top operand stack entry, converts it to base 10, places it into the string (*cvrs*) and then paints the string at the current cursor position on the page (*show*).

Line 7. *sqr* is defined as a function which duplicates the top stack entry and multiplies the two top entries together (i.e., squares the top stack entry).

Line 8. Function *dosum* takes the operands n and S and produces $n+1$ and $S+(n+1)^2$. (See explanation in Figure 5.1 of this *IG*. *exch* is used instead of Forth's *swap* and 1 *index* replaces Forth's *over*.))

Line 9. Try *dosum* with arguments 3 and 6 producing 4 and 22. Make a copy of the 4 and 22 and print 22, print a string of 3 spaces, and then print 4 at the current cursor position.

Line 10. Clear the stack of the remaining 4 and 22.

Line 11. Move cursor down about one-third of an inch (25 points).

Line 12. 0 1 9 *proc for* will place 0, 1, 2, ..., 9 on the top of the stack and then evaluate *proc*. On each iteration, *pop* (to get rid of the index) and *dosum* are called, letting operand stack become: (0,0) $\Rightarrow$ (1,1) $\Rightarrow$ (2,5) $\Rightarrow$ (3,14) $\Rightarrow$ (4,30) ... (10,385). The top of the stack (385) is then painted on the page (*formatit*).

Line 13. The painted page is then printed.

Sample Postscript operators. For each operator the number of arguments (top stack entries) needed to execute the operator and the number of resulting values pushed on the stack are indicated. The usual binary operator (such as *add*), which takes 2 arguments and replaces them with a single result, is given as (2,1).

add (2,1) – *Add* takes the top two entries and replaces them by their sum.

aload (1,n+1) – Place the elements of the array argument onto the stack below the array.

array (1,1) – Create an array of length equal to argument.

astore (n+1,1) – Store n elements into top array argument.

Boolean (2,1) – Compares the top two operand entries and sets the top of stack to true or false. *Boolean* can be *eq*, *ne*, *ge*, *gt*, *le*, or *lt*. In addition, there are operators for *and*, *or*, *not*, *xor*, with their usual interpretation.

clear (n,0) – Pop all entries off the stack.

cleartomark (n,0) – Pop all elements down to the mark on the stack.

copy (n+1,2n) – The n stack entries after the top value of n are duplicated on the stack.

counttomark (n,n+1) – Count the number of arguments from the mark and place value on stack.

currentfont (0,1) – Place current font on stack. (Can use this to resize current font without knowing what the current font is, e.g., currentfont 2 scalefont setfont will double to size of the current font.)

cvrs (3,1) – a b c cvrs stores number a in base b into string c and returns the substring containing a. Thus 123 10 10 *string* returns the string (123). 123 2 10 *string* will convert 123 and return it as a string of binary bits.

def (2,0) – /a b def will take literal name a and function definition b and create an entry for a in *userdict*. Usually, b is given as a list of operations $\{x\ y\ z\}$.

div (2,1) – *Div* takes the top two entries and replaces them by their quotient.

dup (1,2) – *Dup* takes the top stack entry and duplicates it on stack.

exch (2,2) – *Exch* exchanges the position of the two two stack elements.

findfont (1,1) – Take name and place in the stack the appropriate font from the system dictionary.

for (4,a) – a b c proc for places a, then $a+b$, then $a+2b$, ... until c on the stack and then invokes *proc*. The index argument remains on the stack unless *proc* removes it.

forall (2,a) – s proc forall invokes *proc* for each element of string s.

if (2,0) b proc if executes *proc* if b is true.

ifelse (3,0) b proc1 proc2 ifelse executes *proc*1 if b is true and *proc*2 is b is false.

index (n+2,n+3) – The sequence $a\ b\ c\ \ldots d\ n\ index$ will take the n^{th} stack element and duplicate it on the top of the stack. $n = 0$ is the current top element, $n = 1$ is the next, and so on. *Dup* can be replaced by 0 *index*.

length (1,2) – Place the number of elements in array argument onto the top of the stack.

mark (0,1) – Place mark on stack.

moveto (2,0) – The two arguments represent the x and y coordinates from the lower left corner of the page. Size is represented in points, 72 to the inch.

mul (2,1) – *Mul* takes the top two entries and replaces them by their product.

pop (1,0) – This removes the top stack element.

quit (0,0) – This terminates the execution of Postscript.

repeat (2.1) – n proc repeat causes *proc* to be executes n times.

rmoveto (2,0) – This sets the new position relative to the current position.

roll (n+2,n) – a b c ... d m n roll causes the top m objects to be "rolled" circularly n positions.

scalefont (2,1) – Take the font and a scale factor and replace by an operand font entry that represents the appropriately sized font.

setfont (1,0) – Take the scaled font entry and make it the current font to be used in printing.

show (1,0) – Place top stack entry and place it on page.

showpage (0,0) – Print current page and reset page to blank.

string (1,1) – Creates a string consisting of zeros, the length of which is the argument (e.g., 10 string creates a string of 10 zeros.)

sub (2,1) – *Sub* takes the top two entries and replaces them by their difference.

```
14: % Lets draw a truck
15: /box {newpath 0 0 moveto 0 1 lineto 3 1 lineto 3 0 lineto
16:     closepath} def
17: .1 setlinewidth 0 setgray
18: gsave
19: 72 72 scale
20: 2 5 translate box stroke
21: 3.2 0 translate .5 .5 scale box fill
22: 0 1 translate .6 .6 scale box fill
23: grestore
24: /tire {newpath 1 0 moveto 0 0 1 0 360 arc closepath} def
25: .5 setlinewidth 10 10 scale
26: 16 34 translate tire stroke
27: 3 0 translate tire stroke
28: 17 0 translate tire stroke
29: 3 0 translate tire stroke
30: 8 0 translate tire stroke
13: showpage
```

Figure 5.4. Postscript picture.

5.2.3 Painting commands

So far we have described a language very much like Forth. Operands are stacked and operators use stack contents for their arguments. What gives Postscript is power is the ability to paint information on the page.

Some of this we have already seen. *Show* is used to place text onto the page, *moveto* moves the cursor around the page, and *showpage* is used to display a completed page. Here we extend this concept to be able to draw more extensive pictures.

The page is assumed to be white. Objects drawn on the page are opaque, meaning any object will erase an object it overwrites. Erasing part of a page is simply to write a white object onto that space.

The origin is the lower left bottom of the page and locations are given as (x, y) coordinates, with x being the horizontal distance and y being the vertical distance from the origin. Distance is measured in points, 72 to the inch, although the *scale* command can change that.

Example picture. In order to demonstrate some of these graphics capabilities, we will extend the previous Postscript program with a picture enclosing the numbers we previously displayed. Figure 5.4 of this *IG* is our modified program, with lines 14 to 30 inserted between lines 12 and 13 of the previous Postscript program (Figure 5.3 of this *IG*).

Line 14. This is a comment stating we will now draw a picture.

Lines 15–16. This defines a new function *box* that creates a 1 by 3 rectangle. Note that this is 1 point by 3 points – way too small to be seen. We will use the scale command on line 19 to enlarge this. *Newpath* starts a path. The cursor is moved to the origin by the *moveto* command, and then *lineto* is used to create a series of line segments. *Closepath* is used to close the path into a closed figure.

Line 17. Line widths are set to .1 and the color is set to degrees of gray by the *setgray* command. Black is 0; white would be 1.

Line 18. The current graphics environment is saved so we can restore it later. All scaling changes below will be reset to the original value when we restore the current graphics state.

Line 19. We will scale the x and y coordinates so that each point now represents an inch (72 points). A one by three rectangle now represents a one inch by three inch box.

Line 20. We will translate the origin by 2 (inches now, due to scaling) in the x direction and 5 inches in the y direction. The origin (0,0) is now near the numbers printed by lines 1–12 of this program. *box stroke* now draws the 1 by 3 box in the translated location, and stroke fills in the line segments. (We could have also filled in the box with the *fill* command, as we show on the next line.)

Line 21. We now move the origin 3.2 inches in the x direction and 0 inches in the y direction from the previously translated origin. Our new scale of .5 by .5 changes the previous 1 = 1 inch into 1 = .5 inches. *box fill* draws the reduced box and fills it in with black. This represents the lower part of the front truck cab.

Line 22. We further reduce the scale and draw the upper part of the truck cab.

Line 23. Restore the graphics environment so that 1 = 1 point = $\frac{1}{72}$ of an inch.

Line 24. Define a tire as a unit circle. Note that the *moveto* command moves to a point on the circle so that the *arc* command doesn't draw a line segment to the initial point on the circle.

Line 25. Reset the tire width and tire size via the *setlinewidth* and *scale* commands. Tires will be $\frac{10}{72}$ of an inch.

Line 26–30. The 5 tires are drawn by successively moving (via *translate*) the origin to the right.

The complete program is given in Figure 5.5 of this *IG* and the completed picture is given in Figure 5.6 of this *IG*. Note that we added the comment line %! at the beginning. This is a Postscript convention (see Section 5.2.4 of this *IG*) that is often needed to inform programs, such as printing programs, that the file really represents a Postscript program.

Space filling commands. One of the more interesting properties of Postscript is the ability to fill shapes with a pattern. We already used the *box fill* command on line 21 of our truck picture. Given an arbitrary shape, when is one inside or outside the shape? Two algorithms are implemented:

```
%!  This is a Postscript file
%Same as Forth program on page 248 of text
/Helvetica findfont
20 scalefont
setfont
200 400 moveto
/formatit {10 10 string cvrs show} def
/sqr {dup mul} def
/dosum {exch 1 add exch 1 index sqr add} def
3 6 dosum 2 copy formatit ( ) show formatit
clear
200 375 moveto
0 0 0 1 9 {pop dosum} for formatit
% Lets draw a truck
/box {newpath 0 0 moveto 0 1 lineto 3 1 lineto 3 0 lineto
    closepath} def
.1 setlinewidth 0 setgray
gsave
72 72 scale
2 5 translate box stroke
3.2 0 translate .5 .5 scale box fill
0 1 translate .6 .6 scale box fill
grestore
/tire {newpath 1 0 moveto 0 0 1 0 360 arc closepath} def
.5 setlinewidth 10 10 scale
16 34 translate tire stroke
3 0 translate tire stroke
17 0 translate tire stroke
3 0 translate tire stroke
8 0 translate tire stroke
showpage
```

Figure 5.5. "18 wheeler" Truck program.

- *Non-zero winding rule.* This is the method used by the *fill* operator. Draw a line from a point to infinity. For each segment of the shape boundary that crosses the line segment from left to right add one, and for each segment that crosses from right to left, subtract one. If the final count is zero then the point is outside the shape, otherwise it is inside.

- *Even-odd rule.* This is the algorithm used by the *eofill* operator. Draw a line segment from the point to infinity. If it crosses an odd number of shape boundary segments, the point is inside the shape; if an even number of

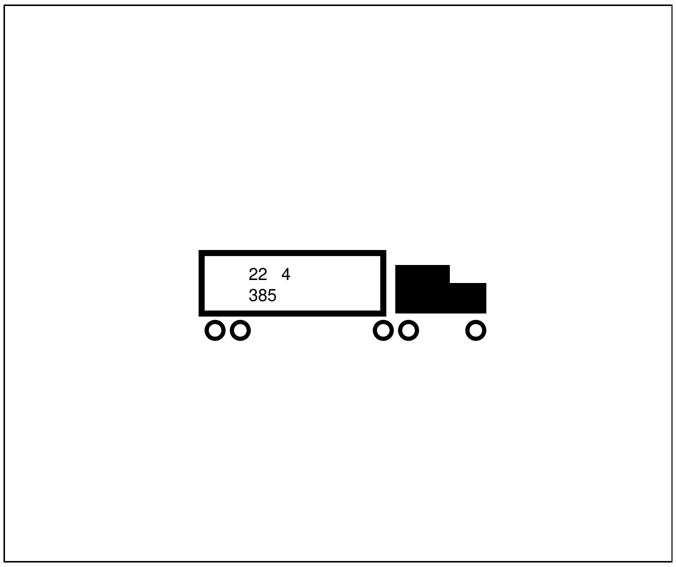

Figure 5.6. Excellent art work by the author.

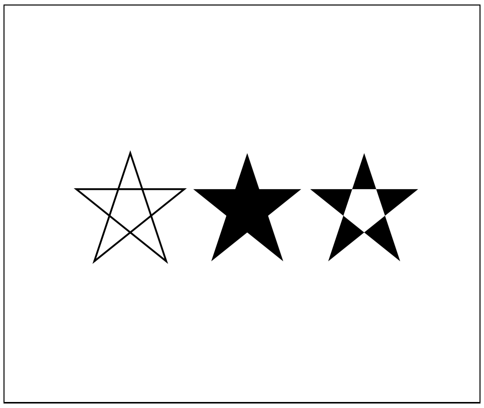

Figure 5.7. Stroke, fill, and eofill commands.

segments, then it is outside the segment.

The differences among filling operators is given in Figure 5.7 of this *IG*. The star is defined by the procedure:

```
/star {newpath 1 0 moveto 6 4 lineto 0 4 lineto 5 0 lineto
        3 6 lineto closepath} def
```

The leftmost start is drawn by the commands *star stroke*, the center star by *star fill*, and the right star by *star eofill*.

Environment operators. These save and restore the state of the Postscript interpreter.

save and restore (0,1) and (1,0) – *Save* places a copy of the current environment on the stack, and *restore* restores the saved environment. This is useful, for example, if scaling and translation are later applied to the environment and you desire to go back to the previous state. These commands also save the graphics state.

gsave and grestore (0,1) and (1,0) – *gsave* saves the current graphics state and *grestore* restores it as the current state.

Painting operators. These commands are useful for painting pictures on the page.

arc (5,0) – x y r a b arc draws an arc with center coordinates x and y, radius r from angle a (in degrees) to angle b. For example, to draw a circle centered at coordinates x and y, do:

```
x y moveto
z y r 0 360 arc
```

where z is $x + r$

closepath (0,0) – This closes the path back to the previous *newpath* command.

currentpoint (0,2) – This places the current coordinates on the stack.

eofill (0,0) – This will paint the current path with the fill color (see *setgray*). To erase an area, set the fill color to white. This operator uses the even-odd rule for strange-shaped objects.

erasepage (0,0) – Paint page white (i.e., erase all contents of page).

fill (0,0) – This will paint the current path with the fill color (see *setgray*). To erase an area, set the fill color to white.

lineto (2,0) – This draws a line from the previous point to this new point.

newpath (0,0) – Sets the current location as a new path.

scale (2,0) – Scale the new coordinate system relative to the old coordinate system (i.e., multiply each x and y coordinate by the arguments to *scale*.

setdash (2,0) – a b setdash sets the pattern for producing line segments. a is an array, whose elements are accessed circularly and specify the sequence of solid and blank line segments. b specifies the distance into the array to start, measured in line distances not array indices. Some examples:

[ ] 0 setdash	A solid line
[1] 0 setdash	One unit solid, one unit blank, one unit solid, ...
[1] 1 setdash	Same as above, only start with blank segment
[1 2 3] 2 setdash	1 unit blank, 3 solid, 1 blank, 2 solid, 3 blank, ...

setgray (1,0) – Sets current color as shades of gray. 0 is black and 1 is white.

setlinecap (1,0) – Determines how line segments are joined in a path, 0 is butt-ended, 1 is rounded, 2 is square.

setlinewidth (1,0) – Sets current width of line.

stroke (0,0) – Draw a line along current path.

translate (2,0) – Translate the user space by relative distances given by the arguments. This changes to location of the origin from the lower left corner to elsewhere on the page (if arguments are positive).

5.2.4 Document conventions

What has been described so far is enough Postscript in which to write programs that create interesting documents. However, there are some conventions that are often used and are sometimes required by various Postscript processors. These are generally comments (which can be ignored), but often convey information to the Postscript interpreter. This prologue will be at the beginning of the file.

The most important is that the leading comment should be:

%!

This informs the interpreter that the file is a Postscript program. Most programs that generate Postscript place their name on this line. A minimally conforming program contains %!PS in its initial line.

Each page of a document is usually bracketed by a *save* and a *restore* command to isolate that page from the effects of other pages.

Structuring conventions begin with %% until the first line that doesn't contain this comment. Some of these conventions are:

DocumentFonts: is a list of fonts used in the document.

Title: Is an arbitrary string giving the title of the document.

Creator: is the name of the person or program that created the file.

CreationDate: is the date and time of creation.

Pages: is the number of pages in the document.

BoundingBox: is the 4 values that represent the lower left and upper right corners of the page that are actually painted by the program. This allows the pages to be inserted into other documents.

5.3 The World Wide Web

The growth of the Internet and the activity of "surfing the net" has led to a growth in the interest in the static description languages mentioned at the beginning of this chapter. In this section was developed and where it came from, and provide a brief overview Java, as an extension to the ideas in HTML, so we will explore some of the details of Java.

5.3.1 The Internet

During the 1960s, computers moved away from the batch-oriented model of computing toward an interactive model using terminals (See Section 1.4 of the text). As machines became more powerful, there was a desire to share information among users on different systems.

In the late 1960s, the U.S. Department of Defense Advanced Research Projects Agency (ARPA) began a project to see if several computers, widely separated geographically, could be linked together so that users at a terminal on one system could access the resources (e.g., the programs and data) on another computer. This was the beginning of the ARPANET. It is beyond the scope of this book to cover this in great detail. Suffice it to say that data communications, sending messages reliably from one computer to another, was the major obstacle.

The initial ARPANET began in 1970 as a three node network linking BBN in Cambridge, Massachusetts with UCLA and SRI in California using 56 kilobit lines. Gradually over time more military installations and universities were added to the ARPANET until several hundred sites were available by the mid-1970s.

Communication between two computers was handled via *messages* (See page 396 of text). In order to make sure that messages destined for another computer arrived there reliably, a formal communication model, called a *protocol*, was developed. For the ARPANET, this became known as TCP/IP (Transmission Control Program/ Internet Protocol).

TCP/IP was a low level communication mechanism that simply determined that a sequence of bytes destined to a specific computer arrived there uncorrupted. It was generally too complex to use for normal computer use. Therefore, several other protocols were developed that users could invoke, and these new protocols would be implemented in programs that actually transmitted the data using TCP/IP. For the ARPANET, three protocols became widely used:

- *Telnet.* Telnet is a protocol that makes the computer the user is actually working at look like a terminal to the distant computer. (See Section 9.6.2 of text.) Using the terminology of distributed computing, the user is connected to a client computer, which acts like a terminal, and the terminal program is communicating using the telnet protocol to a distant *host* computer which is providing the server program. This allows a user at one location to log into a computer at another location and appear to the distant computer as a local user.

- *SMTP.* SMTP is Simple Mail Transport Protocol. This provides the basic e-mail (electronic mail) that has become so ubiquitous today.

- *FTP.* FTP is File Transfer Protocol. For many years this was the standard way to retrieve data from a distant machine. One would invoke the FTP client on a local machine, log onto the distant machine using the FTP protocol, and then retrieve the desired documents.

Weaknesses in the FTP mechanism are apparent. First of all, one had to know explicitly which machine to access in order to retrieve the desired data. One also had to have access to the files of that machine in order to retrieve the information. This was handled somewhat by the concept of an "anonymous" user that anyone could use to access a restricted set of files. Also, one had to know exactly where on the file system the desired information was.

In spite of these weaknesses, FTP was the standard file transmission mechanism for many years, until HTML changed all that. But more about HTML later.

In the mid-1980s, ARPA decided to rid itself of the ARPANET. By this time it had grown to several thousand computers worldwide. The academic world, the military, and high-technology companies were heavy users of this system. The U.S. National Science Foundation (NSF) took over the "backbone" network in the U.S., and the name gradually evolved into the Internet. Attached to this backbone, local networks (a state, a university, a large company) were added, until the Internet became this amorphous collection of computers, all chattering to one another continually. Commercial providers (e.g., Compuserve, AOL) established links to the Internet until today no one can accurately determine how many millions of users actually have access to this network.

By the early 1990s, the NSF also wanted to get out of the networking business, and various commercial communications companies took over management of the network. Growth has continued until today there are an estimated 500,000 computers and millions of users. The actual numbers are impossible to determine due to the lack of any centralized control of Internet resources. Transmission speeds have increased from the initial 56 kilobit lines, to 1 megabit lines to 100 million bits per second on fiber optics cables, although individual users working at home typically are connected via 14.4 kilobit or 28.8 kilobit per second modems.

5.3.2 HTML

The World Wide Web (WWW) was developed as an additional protocol to the three (Telnet, SMTP, and FTP) mentioned above. As stated above, FTP had its problems. Physicists, principally Tim Berners-Lee, at CERN, the high energy particle research laboratory near Geneva, Switzerland, desired a mechanism to access and transfer documents by computer that was simpler than the standard FTP server. They developed the concept of the static description given earlier in this chapter. One server program would display a document, and a client program, called a *browser*, would read and understand the displayed program. The power of their system was that the displayed document contained pointers to other documents, something the computer science community calls *hypertext*. An earlier version of hypertext was Apple Computer's HyperCard product for the Macintosh, but the real power of the CERN development was to allow hypertext links to documents that existed on other computers connected to the Internet.

The protocol developed was the HyperText Transport Protocol or HTTP. Each pointer became known as a URL or *Uniform Resource Locator*. Document location reduced to invoking a WWW browser on your local machine, connecting to a WWW server on a distant machine, displaying a document obeying the HTTP protocol, and "clicking" with your mouse on URLs embedded within the displayed document as hypertext links to locate other documents.

This was vastly superior to the FTP mechanism that was used earlier. WWW browsers totally changed the nature of the Internet. The National Center for Supercomputer Applications (NCSA) Mosaic browser (since commercialized by Spyglass) was an early favorite, while the Netscape Navigator browser is heavily used today. As long as a browser obeys the HTTP protocol, any browser can communicate with any WWW server. This moved the Internet out of the realm of the "techie" user and into mainstream computer (and non-computer) applications.

As for the actual details of the notation for creating HTTP, CERN decided to use a version of SGML (Standard Generalized Markup Language) that they called HyperText Markup Language (HTML). We will first look at the general SGML notation and then look at the HTML instance of SGML.

SGML

History of SGML. Development of SGML (Standard Generalized Markup Language) began in the late 1960s as a means to format a particular document using descriptive tags. Rather than using proprietary keywords, a document could be classified according to a predefined set of terms, such as "section heading," "title," "figure," etc. In the late 1960s, the Graphic Communications Association Composition Committee, with ideas from William Tunnicliffe, Stanley Rice, and Norman Scharpf, developed the GenCode(R) concept to identify the different generic codes needed for different documents.

In 1969, Charles Goldfarb of IBM with Edward Mosher and Raymond Lorie developed the Generalized Markup Language (GML) to identify components of a document with a specific nested syntax.

In 1978, the American National Standards Institute (ANSI) set up a standards group to merge the concepts of GenCode(R) and GML. This developed into SGML, which was standardized in 1986.

SGML syntax. A document consists of text, which is an unstructured sequence of characters. However, within the text can be SGML *elements*. The semantics of elements are unspecified, but their syntax is given. Elements are bracketed by a start-tag and an end-tag notation. For example, the *zork* element begins with the tag ⟨zork⟩ and terminates with the end tag ⟨/zork⟩. All text between these tags belongs to this element:

⟨zork⟩ I am a zork. ⟨/zork⟩

identifies "I am a zork." as the contents of the *zork* element.

In cases where the ending of an element is unambiguous, the end-tag can be omitted. For example, if we define a report as an SGML document consisting of a title, author, abstract, and body of text, we could give its description as:

⟨report⟩
⟨title⟩ text ⟨/title⟩
⟨author⟩ text ⟨/author⟩
⟨abstract⟩ text ⟨/abstract⟩
⟨body⟩ text ⟨/body⟩
⟨/report⟩

or as:

⟨report⟩
⟨title⟩ text
⟨author⟩ text
⟨abstract⟩ text
⟨body⟩ text
⟨/report⟩

since each section flows unambiguously into the next.

If we define the body of a report to be a sequence of sections, each consisting of nested sections, to an unlimited depth, then each section is defined by the syntax:

⟨section⟩ text ⟨/section⟩

but in this case, the end-tag would be required since it would not be unambiguously known where the end occurred.

The use of SGML separates the attributes of a document from its presentation. We could print the above ⟨report⟩ using any type font and not destroy the essential meaning of the document. We could print the ⟨title⟩ element in 20 point font and the ⟨author⟩ element in 16 point font, or we could print everything in 10 point font. All this is outside of the scope of the document itself.

We could also use this to handle more significant presentation issues. As given in Section 4.2.5 of the text, the date is often displayed in different formats in different countries. To write that "There will be a meeting on 11/10/96." has a very different meaning in the U.S. and Europe. If this is an internal memo of an international company, it would not be clear if the meeting is on the 10^{th} of November or the 11^{th} of October. However, if this memo were in SGML, we could have an appropriate ⟨date⟩ element and write the statement as:

```
⟨memo⟩ There will be a meeting on
    ⟨! This is a SGML comment -- Give date⟩
    ⟨date⟩ ⟨month⟩ 11 ⟨day⟩ 10 ⟨year⟩ 96 ⟨/date⟩.
⟨/memo⟩
```

and different client mail systems, one for the European employees and one for the U.S. employees, would display the ⟨date⟩ element correctly for each user.

The set of elements are described in an SGML component called the DTD (Document Type Definition), which provides for a grammatical definition of the available tags for a given document.

While there are other components of SGML, this is a sufficient introduction for our tour of HTML.

HTML Syntax and Semantics

When CERN developed the initial concept of a hypertext browser, SGML became the basis for their notation. The language was called HTML (HyperText Markup Language) and most files are given as *name.html* by convention. (On PCs, which have three character filename extensions, this is often given as *name.htm*.) The default file to read when a WWW browser tries to access a new computer is usually the name *index.html*. For example, if you want to access updated information about *Programming Languages: Design and Implementation (Third Edition)*, this author has set up a WWW "home page" at the University of Maryland. By entering the address:

http://www.cs.umd.edu/users/mvz/pzbook/

your WWW browser will look for the file *index.html* in this directory, and then display it. (The meaning of the syntax *http* ... will be given shortly.)

HTML files obey the SGML syntax given previously. The minimal HTML document has the syntax:

```
⟨html⟩
⟨title⟩ Document title ⟨/title⟩
⟨body⟩ Text of document ⟨/body⟩
⟨/html⟩
```

Case is ignored for tags. Thus ⟨html⟩ and ⟨HTML⟩ are the same. In addition, tags can sometimes have optional arguments, as given below.

The text of the body of the document can have additional structure. Some of the more important elements of HTML are as follows:

Sections. These separate parts of a document.
⟨h1⟩ Major heading ⟨/h1⟩. These are usually printed in large letters.

⟨h2⟩ Minor heading ⟨/h2⟩. These are usually printed in large letters, but smaller than ⟨h1⟩ letters.

⟨hr⟩. Draw a horizontal line across document. The tag width (e.g., ⟨hr width = 50%⟩) specifies how long the line should be.

⟨p⟩. Start a new paragraph. (No ⟨/p⟩ is needed.)

⟨br⟩. Start next line (with no extra space as in ⟨p⟩).

Presentation. These affect how a document may look. Exactly how they are displayed is not specified, and individual browsers have wide latitude in presentation format.
⟨b⟩ text ⟨/b⟩ displays text in bold fonts.

⟨i⟩ text ⟨/i⟩ displays text in italics.

⟨blink⟩ text ⟨/blink⟩ displays blinking text (which is very annoying and should not be used).

⟨pre⟩ text ⟨/pre⟩ is pre-formatted text. Display this as it exists in the HTML file with all embedded spaces and newline characters. In general, HTML ignores "whitespace" characters as it formats the text for presentation.

⟨center⟩ text ⟨/center⟩ centers the text.

⟨blockquote⟩ text ⟨/blockquote⟩ formats a long quotation.

Lists. Lists of items can be displayed in various ways:
⟨li⟩ starts the next item in a list. This is terminated by either the end of the list or the next list item. (There is no ⟨/li⟩.)

⟨ul⟩ list items ⟨/ul⟩ is an unnumbered list of items. Each item begins with the tag ⟨li⟩.

⟨ol⟩ list items ⟨/ol⟩ is a list of numbered items. Each item begins with the tag ⟨li⟩.

⟨dl⟩ definitions ⟨/dl⟩ is a definition list. Each item consists of the tags:
  ⟨dt⟩ term being defined
  ⟨dd⟩ definition of term

URLs. The power of HTML is in its links to other HTML resources. The concept of the Uniform Resource Locator (URL) is the mechanism that provides that. In specific tags of an HTML document, the URL of another HTML document can be accessed. The general syntax of a URL is:

http://location/pathname/file

where *http* : signifies that the location is an HTML document (the HTTP protocol), // states that it is on another machine, and *location* is its address on the Internet. If the URL points to a local file, then the leading // can be omitted.

Accessing another document is given by the tag ⟨a⟩.

⟨a HREF = url location⟩ text ⟨/a⟩ causes *text* to be highlighted in some way by the WWW browser. Clicking the mouse on this text causes the browser to link to this new HTML file at address *url location* and display it. This is the basic mechanism used to "surf the net."

⟨IMG SRC = file.location ALT = text⟩ causes the gif-formatted picture at file location *file.location* to be displayed on the screen. ALT indicates *text* that should be displayed if the browser cannot display pictures. The parameter *align* (with values *top*, *middle*, or *bottom*) causes succeeding text to be aligned with the top line, middle, or bottom line of the figure. This is extremely limiting in that only one line of text can appear next to a picture. (See HTML future, below.)

Applets

As given so far, HTML is a very passive language. The browser simply navigates from one HTML file to another displaying information or pictures. However, WWW servers have additional features that permit a more active role for the server. It can acquire information from the client browser (i.e., from the user) and provide for alternative behavior. We are in effect using the WWW server to execute programs. These have become known as *applets* or mini-applications. Some of these are available on all WWW servers. Sun Microsystems has been developing a new concept, called Java, which we will discuss in the next subsection.

In defining a URL, we have the strange syntax in the HREF option of "http:". This is because an HTML file is only one possible option of a hypertext link. Others are possible. The general syntax is:

method://location/pathname/file

which indicates that the URL given should be accessed via the chosen *method*. HTML files via the HTTP protocol is only one way to create hyperlinks in a document. Others include:

FTP	- Access the URL using the FTP protocol. This is used to transmit files to the browser, especially non-HTML files. HTML files can be easily transmitted directly by the browser.
GOPHER	- Access the URL using the gopher protocol. Gopher is an earlier protocol for accessing information around the net. It was just gaining popularity when HTML arrived on the scene and replaced it.
MAILTO	- The URL in this case is an email address, and the WWW server creates a mail message to that address. This allows the creator of the HTML file to receive mail about the contents of the file.
FILE	- The URL is a local file. Based upon the extension to the file name, an appropriate viewer will be invoked. That is, *report.ps* will open a Postscript viewer for the file, *picture.tif* will open a graphics view in TIFF format, *sound.au* represents an audio file, etc.
NEWS	- The URL of *news.group* will invoke the local news server to access that news group.

One of the weaknesses of the WWW is that each browser has to know about each of these protocols. If a new protocol is developed, then each browser needs to be updated to handle this new feature. Given the amorphous and sometimes chaotic growth of the Internet, this is a recipe for disaster on the net. Java was developed to handle this problem, as we discuss later.

Forms. Forms are a method for inserting special commands into an HTML document. Information is entered by the user, based upon embedded commands in the HTML file, and this information is passed to another program on the server system. This is the CGI (Common Gateway Interface) file. WWW servers typically have a specific *cgi-bin* directory. The syntax of the ⟨form⟩ tag is given as:

⟨form method = "type" action = "cgi script to execute"⟩
text ⟨/form⟩

where *type* can be either *get* or *post*. *Get* passes the data as arguments to the CGI script, *post* passes the information through the normal read process of the CGI program. The CGI script is another program. This can be a C program, a shell program (in UNIX), or a TCL or PERL script. (TCL, PERL, and the shell are beyond the scope of this book.)

Several tags provide for the information that can be passed to this CGI program:

⟨input type = "text" name = "name" size = number⟩

which causes (in most browsers) a window of size *number* to open on the screen. The pair (name, user's input) will be passed to the CGI program.

The tag ⟨input type = "submit" value = "send"⟩ creates a button with the label *send*, which when clicked on causes the data entered in the form to be sent to the

CGI program. The tag ⟨input type = “reset” value = “Reset form”⟩ clears all data from the form to allow for reentering the data before submission.

Other types are implemented for the ⟨input⟩ tag. ⟨input type = “password”⟩ is similar to *text* except that the input does not appear on the browser screen. As its name implies, it is useful for typing in passwords. ⟨input type = “checkbox”⟩ creates a box that can be checked (i.e., a toggle button or a push pin icon); ⟨input type = “radio”⟩ creates a radio button for other toggles with the same name.

⟨textarea name = “name” rows = num1 cols = num2⟩ is like ⟨input⟩, except that a box that is $num1 \times num2$ is opened on the browser window.

The data sent by the form will appear to the CGI program as values for environment variables. The environment variable *CONTENT_LENGTH* is set to the length of the submitted data. Thus if *Marvin V. Zelkowitz* is entered in response to the ⟨input⟩ form of type *text*, then the CGI program can access this information by obtaining the value of the environment variable *name*. In this way, information entered by the user on a WWW browser can be passed to a program on the server system to be processed. This information can result in specific information being returned to the user, the data can be stored by the server, or a specific new HTML document can be opened.

A common technique is to create a response to this input by the user by writing an HTML file and then displaying that file via a URL hyperlink. This is the way many search query engines work when you surf the net:

1. A form is used to pass information to the CGI search program.
2. The search program locates the desired information.
3. The search program formats the information as a text file using HTML syntax.
4. The server does a hyperlink transfer to this new HTML file, displaying the answer on the browser's display screen.

Using forms, CGI scripts, and building HTML files dynamically allow the WWW to behave like any other computing environment. One can build query systems, games, and other features that have made surfing the net so popular.

HTML future. As this is being written, there are various proposals to update HTML version 2 to HTML version 3. Extensions in this new version of HTML include the following:

Images. The attributes *height* and *width* have been added to the ⟨img⟩ tag as well as additional directives on the *align* attribute. This permits two important extensions:

- Text can flow around a picture and not only be aligned on the top or bottom line.

- This permits the text of an HTML document to be displayed before the enclosed pictures. One of the problems with the WWW is that pictures can take hundreds of thousands of bytes, while the text on the page may only be a few thousand. This causes long transmission delays while you wait for the pictures to arrive, even if you are only interested in the text. These new options permit faster information transfer by delaying picture transmission until after text.

This is a word of warning to all of you that are building WWW "home pages." Everyone likes to build pages with lots of creative graphics. The problem with graphics is the time the receiver of that information has to wait in order to get the desired information. Be aware of this as you include pictures in your HTML files.

Tables. Tables have been created with a ⟨table⟩ tag. ⟨tr⟩ starts a new table row, ⟨td⟩ is the next item in that row, ⟨th⟩ heading ⟨/th⟩ gives a cell heading, and ⟨caption⟩ text ⟨/caption⟩ places a caption after the table.

Style sheets. The structure of documents will be described via HTML style sheets; however, the design of those style sheets are not well formed as this is written.

5.3.3 Java

The existing WWW structure with HTML documents being read by WWW servers and displayed by WWW browsers has three limitations. It is clear that the web has greatly exceeded the initial design goals developed by the physicists at CERN:

1. For each change or addition to a protocol, all browsers need to be updated. It does no good to just update the server to display the new format, since the display process resides in the browser client.
2. The amount of information an HTML process can access is limited to the few simple keystrokes generated by the ⟨form⟩ elements of a document. It would be useful for HTML programs (e.g., the CGI scripts) to access large data files on the browser system.
3. Control of the interaction is now handled by the server with the embedded HTML commands. This ties up the server. It would be more efficient if control could be managed by the browser with the server being in more of a supporting role.

One way to handle this would be for the server interaction to execute on the browser platform. This is the essential feature of Java. It allows for extensibility since all protocols are determined interactively at run time.

HotJava WWW browser

Java is a programming language, described in more detail later, It has a syntax similar to C++ with some of the semantics of C++ and Smalltalk. Java programs are initially compiled into a machine independent interpreter code. This interpreter code can be executed on any host machine that has the appropriate Java interpreter.

Sun has developed a web browser called HotJava that includes a Java interpreter. HotJava behaves much like existing WWW browsers with a significant difference. Whereas existing browsers need to know about all protocols (e.g., http, ftp, .gif, .au), HotJava knows about none of them. When a hyperlink URL is activated, the method is checked, and if it is not available, HotJava will look for it on the browser system and then on the server system. If found, it will transfer it (if necessary) to the browser client, and then execute it as a local applet.

For example, if HotJava finds an *http* hyperlink, it will first transport the *http* applet, which has been written in Java, and then execute it. Only those servers implementing a method need have a Java applet for it. The browser, in turn, need know nothing about it, only the ability to acquire it if necessary.

This allows for a somewhat orderly growth of WWW functionality. If a company develops a new protocol, it can implement its server with it and supply a Java applet to handle this new method. The HotJava browser, when it finds this new method, can download the applet and then execute it. One feature (or danger, depending upon your point of view) of this approach is that the company can first charge for accessing this new applet and provides for a source of revenue that is currently missing from most WWW access. Of course there are those who say WWW access should not have a charge. Free access probably cannot survive forever, but a reasonable charging scheme needs to be developed. One pitfall to this is that a company may simply use an existing protocol, but change its name in order to stimulate revenue. One hopes that such charades will fail, since if not, the entire Internet may fail. Internet growth and usage has survived principally on the universality of access to Internet resources.

A side effect of this approach is that once the applet is executing on the client system, it has access to the file system of the user. This provides for true interactive programming with data supplied by the user. Of course, an important consideration is security of the user system. One must be extremely careful that the browser that accesses an applet does not permit it to contain a computer virus (the ability to change executable programs on the user's machine to destroy files) or a Trojan horse (the ability to take over control of the user's computer). Certification of such browsers is going to be of critical importance before they are widely distributed.

The Java language.

Java has a syntax much like C++ with many of the problem areas of C and C++ deleted. It is strongly typed with numbers, Boolean, and characters all being different data types. On the other hand, it does not have pointer data or user access to functions like *malloc*, and thus avoids many of the problems with C storage management.

A Java program consists of a set of class definitions, each containing several method definitions. The syntax should be easy to follow if you have studied C++ (Section 12.2 of text).

Hello World. Our ubiquitous Hello World program can be written in Java as follows:

```
class HelloWorldProgram {
    static public void main(String args[ ]) {
        System.out.println("Hello World!");
    } }
```

The applet declares a class named *HelloWorldProgram* consisting of the single method *main*. *args* represent command line arguments to the program. The one statement of *main* is the passing of the string "Hello World!" to the *println* method of object *out* in the *System* class. That is, the string is printed.

Primitive data. Numeric data can be 8-bit, 16-bit, 32-bit or 64-bit signed integer data. There is also 32-bit and 64-bit real data that follows IEEE standard 754 (Page 135 of text).

Character data is stored using *unicode*. Unicode is an international standard that creates 16-bit characters.

```
char x = 'a'
```

creates an object x with the unicode value for the character a.

Boolean data can have the values of *true* or *false*. Unlike C, Boolean is not an integer value.

Structured Data Objects. Arrays are explicitly defined in the language:

```
Newvalues Mydata[ ]
```

declares *Mydata* as an array of *Newvalues*, where *Newvalues* is a class object you define. However, to actually allocate the array, you would invoke the *new* function, as in the following to allocate a 100 element array:

```
Mydata = new Newvalues[100];
```

To actually allocate each element of the array, you would also need to do, for each i:

```
Mydata[i] = new Newvalues()
```

The *length*() method to array objects returns the length of the array:

```
Mydata.length()
```

will return 100 in this case. as in C++.

Strings are objects with a syntax similar to C:

String Publisher = "Prentice Hall";

instantiates a string object initialized to "Prentice Hall". The + operator for strings means concatenation:

System.out.println(Publisher + " is the publisher of this Guide.");

will print out "Prentice Hall is the publisher of this Guide." *length*() is also defined as a method for string objects.

There is no need for C *struct* data since the C++ class structure permits composite data by simply defining a class containing the several defined objects.

Statements. Java has the usual C control structures: **if**, **for**, **while**. There is no **goto** statement, You exit loops by the C **continue** or **break** statement by placing a label on the loop to exit:

```
OuterLoop: for(i=0, i<j, i++) {
    ...
    if (true exit condition){break OuterLoop;}
    ...
}
```

Class definitions. Java class definitions are similar to C++ class definitions in defining instance variables and methods for a new class, but have aspects of Smalltalk in that the superclass of a new class is explicitly given, as in:

```
class Newvalues extends Object {
    public double x; /* instance variable */
    public int y; /* instance variable */
    Newvalues() { ... } /* Constructor for Newvalues */
    }
```

where *Newvalue* is a subclass of the superclass *Object*, and consists of the two instance variables x (a double real value) and y (an integer).

As given above, C-like declarations are used to allocate instances of this class object:

Newvalues Mydata[ ]

which creates a (blank) array *Mydata*, each of whose elements are of type *Newvalues*. The actual allocation of storage for the arrays is via:

Mydata[17] = new Newvalues();

which allocates storage for element 17 of the array.

As with C and C++, the same selector syntax is used to refer to the components of object *Mydata*: *Mydata*[17].*x* and *Mydata*[17].*y*.

Also the same as C++, the method *Newvalues*() can be defined to give the instance variables x and y explicit values on instantiation.

We can pass parameters to the constructor as in:

```
Newvalues(double a; int b) {
    this.x = a;
    this.y = b; }
```

A method to be invoked when an object is deallocated is the *finalize*() method (similar to C++'s $\sim$ *classname* destructor).

Other methods can be specified in a class definition and invoked the same way as in C++.

Java has no explicit function call or even a need for one. Method access (as in *out.println*(*string*)) is an implicit function invocation.

Subclass definitions. Java has the same sub-classing structure of C++. A subclass *Newclass* is a subclass of *Olderclass* by the class definition:

```
class Newclass extends Olderclass { ... }
```

Newclass inherits all the instance variables of *Olderclass*, and method invocation to *Newclass* (as in *Newclass.functionX*) is passed to *Olderclass.functionX* if *functionX* is not a method defined in class *Newclass*.

Access control. Instance variables in Java are public, private, and protected. Public names are available everywhere, private names are accessible only in that class definition, and protected are available in subclasses of that class.

If none of those access control names are used, the access control is assumed to be *friendly*, or available to any class definition in the same package as the given class, but not elsewhere (a form of semi-public).

Variables in a class definition are usually *instance* variables, meaning that they are in every instance of an object of that class. To have a variable that is common across all instances of a given class, use the attribute *static*. For example, if you are creating a linked list structure, each linked list would be of class *LinkedObject*. There would be a declaration:

```
static FreeListType FreeList;
```

That would be common across all instances of a *LinkedObject* to manage the allocation of new list elements from a common list. Methods common across all instances of a class and able to access class variables only can also be declared with the attribute *static*.

C++'s virtual methods are called *abstract* in Java, and are defined similarly to C++'s virtual methods:

abstract void MethodName(); /* Null body */

Threads. Java allows for some level of multiprocessing through threads. For example, the HotJava browser has a garbage collector executing in parallel with the browser that is displaying the user's HTML files. The avoids halting displays while waiting for the garbage collection to finish. Also, a method to display a graphic can be a separate thread from one displaying text, allowing for more rapid transfer of useful information.

Threads are defined by the *new Thread(object)* command. The *synchronized* attribute (*synchronized void startSort()*) on multiple method definitions prevents more than one of them from executing at a time, and thus avoids deadlock situations.

Language evaluation

For the most part, Java is C++ with the excess baggage of C, which yields poor programs, removed. Data is strongly typed with integers, Booleans, and characters all being separate types. Arrays are also separate types, and a string is not simply an array of characters. All of these permit the compiler to perform more complete error checking on Java programs.

There are fewer ways to perform some actions, such as method invocation being the only subroutine linkage. Since all objects are part of classes, there is no need for separate function or procedure calls. *Struct* objects are also not needed since the same effect can be achieved via instance variables in class definitions.

Pointers, including all of the problems with memory fragmentation, dangling references, and other pointer mishaps, are hidden from the users. Pointer allocation is implicit by the instantiation of class objects and the *new* operator.

Java is basically a simple language, which has the clean C++ design without some of C's clumsiness. However, this is at a cost in execution efficiency since many of these structures need runtime validation. In addition, since Java is interpreted, there is certainly an execution cost over compiled machine language. But for the application domain it was designed for, WWW browsers, workstation speed is higher than either the terminal display processor or Internet communication. Therefore, the inefficiency of the Java execution model will be mostly hidden from the user, who will still be spending most of the time waiting for network response to transmit the desired information.

As stated earlier, the real key to Java's usefulness is the HotJava (and other) WWW browsers. Security of these browsers is crucial for their acceptance. It must not be possible for a browser to download a malicious Java applet from another source, which then destroys the client's file system. HotJava is supposed to have runtime checks to ensure that this doesn't happen. However, this is really a systems validation issue and outside of the realm of the Java language. Java itself is a programming language somewhat akin to Smalltalk using a C++ notation.

Index