# The Evolution of Programming Languages

## Course Notes for COMP 348 and COMP 6411

These notes may be copied for students who are taking either COMP 348 *Principles of Programming Languages* or COMP 6411 *Comparative Study of Programming Languages*.

**First draft:** August 1999
**Revised:** August 2002

Department of Computer Science
Concordia University
Montreal, Quebec

# Contents

## List of Figures

# 1 Introduction

What is this course about? “Evolution” sounds like history, but this is not a history course.

In order to understand why programming languages (PLs) are as they are today, and to predict how they might develop in the future, we need to know something about how they evolved. We consider early languages, but the main focus of the course is on contemporary and evolving PLs.

The treatment is fairly abstract. We not discuss details of syntax, and we try to avoid unproductive “language wars”. A useful epigram for the course is the following remark by Dennis Ritchie:

> C and Pascal are the same language.

We can summarize the goal of the course as follows: if it is successful, it should help us to find useful answers to questions such as these.

- What is a good programming language?
- How should we choose an appropriate language for a particular task?
- How should we approach the problem of designing a programming language?

Some important topics, such as concurrency, are omitted due to time constraints.

## 1.1 How important are programming languages?

As the focus of practitioners and researchers has moved from programming to large-scale software development, PLs no longer seem so central to Computer Science (CS). Brooks (1987) and others have argued that, because coding occupies only a small fraction of the total time required for project development, developments in programming languages cannot lead to dramatic increases in software productivity.

The assumption behind this reasoning is that the choice of PL affects only the coding phase of the project. For example, if coding requires 15% of total project time, we could deduce that reducing coding time to zero would reduce the total project time by only 15%. Maintenance, however, is widely acknowledged to be a major component of software development. Estimates of maintenance time range from “at last 50%” (Lientz and Swanson 1980) to “between 65% and 75%” (McKee 1984) of the total time. Since the choice of PL can influence the ease of maintenance, these figures suggest that the PL might have a significant effect on total development time.

Another factor that we should consider in studying PLs is their ubiquity: PLs are everywhere.

- The “obvious” PLs are the major implementation languages: Ada, C++, COBOL, FORTRAN, . . . .
- PLs for the World Wide Web: Java, Javascript, . . . .
- Scripting PLs: Javascript, Perl, Tcl/Tk, . . . .
- “Hidden” languages: spreadsheets, macro languages, input for complex applications, . . . .

The following scenario has occurred often in the history of programming. Developers realize that an application requires a format for expressing input data. The format increases in complexity until it becomes a miniature programming language. By the time that the developers realize this, it is too late for them to redesign the format, even if they had principles that they could use to help them. Consequently, the notation develops into a programming language with many of the bad features of old, long-since rejected programming languages.

There is an unfortunate tendency in Computer Science to re-invent language features without carefully studying previous work. Brinch Hansen (1999) points out that, although safe and provably correct methods of programming concurrent processes were developed by himself and Hoare in the mid-seventies, Java does not make use of these methods and is insecure.

We conclude that PLs ***are*** important. However, their importance is often under-estimated, sometimes with unfortunate consequences.

## 1.2 Features of the Course

We will review a number of PLs during the course, but the coverage is very uneven. The time devoted to a PL depends on its relative importance in the evolution of PLs rather than the extent to which it was used or is used. For example, C and C++ are both widely-used languages but, since they did not contribute profound ideas, they are discussed briefly.

With hindsight, we can recognize the mistakes and omissions of language designers and criticize their PLs accordingly. These criticisms do not imply that the designers were stupid or incompetent. On the contrary, we have chosen to study PLs whose designers had deep insights and made profound contributions. Given the state of the art at the time when they designed the PLs, however, the designers could hardly be expected to foresee the understanding that would come only with another twenty or thirty years of experience.

We can echo Perlis (1978, pages 88–91) in saying:

> [Algol 60] was more racehorse than camel . . . . a rounded work of art . . . . Rarely has a construction so useful and elegant emerged as the output of a committee of 13 meeting for about 8 days . . . . Algol deserves our affection and appreciation.

and yet

> [Algol 60] was a noble `begin` but never intended to be a satisfactory `end`.

The approach of the course is conceptual rather than technical. We do not dwell on fine lexical or syntactic distinctions, semantic details, or legalistic issues. Instead, we attempt to focus on the deep and influential ideas that have developed with PLs.

Exercises are provided mainly to provoke thought. In many cases, the solution to an exercise can be found (implicitly!) elsewhere in these notes. If we spend time to think about a problem, we are better prepared to understand and appreciate its solution later. Some of the exercises may appear in revised form in the examination at the end of the course.

Many references are provided. The intention is not that you read them all but rather that, if you are interested in a particular aspect of the course, you should be able to find out more about it by reading the indicated material.

## 2 Anomalies

There are many ways of motivating the study of PLs. One way is to look at some anomalies: programs that look as if they ought to work but don't, or programs that look as if they shouldn't work but do.

**Example 1: Assignments.** Most PLs provide an assignment statement of the form `v := E` in which `v` is a variable and `E` is an expression. The expression `E` yields a value. The variable `v` is not necessarily a simple name: it may be an expression such as `a[i]` or `r.f`. In any case, it yields an address into which the value of `E` is stored. In some PLs, we can use a function call in place of `E` but not in place of `v`. For example, we cannot write a statement such as

```
        top(stack) = 6;
```

to replace the top element of a stack. □

**Exercise 1.** Explain why the asymmetry mentioned in Example 1 exists. Can you see any problems that might be encountered in implementing a function like `top`? □

**Example 2: Parameters and Results.** PLs are fussy about the kinds of objects that can be passed as parameters to a function or returned as results by a function. In C, an array can be passed to a function, but only by reference, and a function cannot return an array. In Pascal, arrays can be passed by value or by reference to a function, but a function cannot return an array. Most languages do not allow types to be passed as parameters.

It would be useful, for example, to write a procedure such as

```
        void sort (type t, t a[]) {....}
```

and then call `sort(float, scores)`. □

**Exercise 2.** Suppose you added types as parameters to a language. What other changes would you have to make to the language? □

**Exercise 3.** C++ provides ***templates***. Is this equivalent to having types as parameters? □

**Example 3: Data Structures and Files.** Conceptually, there is not a great deal of difference between a data structure and a file. The important difference is the more or less accidental feature that data structures usually live in memory and files usually live in some kind of external storage medium, such as a disk. Yet PLs treat data structures and files in completely different ways. □

**Exercise 4.** Explain why most PLs treat data structures and files differently. □

**Example 4: Arrays.** In PLs such as C and Pascal, the size of an array is fixed at compile-time. In other PLs, even ones that are older than C and Pascal, such as Algol 60, the size of an array is not fixed until run-time. What difference does this make to the implementor of the PL and the programmers who use it?

Suppose that we have a two-dimensional array `a`. We access a component of `a` by writing `a[i,j]` (except in C, where we have to write `a[i][j]`). We can access a row of the array by writing `a[i]`. Why can't we access a column by writing `a[,j]`?

It is easy to declare rectangular arrays in most PLs. Why is there usually no provision for triangular, symmetric, banded, or sparse arrays? □

**Exercise 5.** Suggest ways of declaring arrays of the kinds mentioned in Example 4. □

**Example 5: Data Structures and Functions.** Pascal and C programs contain declarations for functions and data. Both languages also provide a data aggregation — `record` in Pascal and `struct` in C. A record looks rather like a small program: it contains data declarations, but it cannot contain function declarations. □

**Exercise 6.** Suggest some applications for records with both data and function components. □

**Example 6: Memory Management.** Most C/C++ programmers learn quite quickly that it is not a good idea to write functions like this one.

```
char *money (int amount)
{
  char buffer[16];
  sprintf(buffer, "%d.%d", amount / 100, amount % 100);
  return buffer;
}
```

Writing C functions that return strings is awkward and, although experienced C programmers have ways of avoiding the problem, it remains a defect of both C and C++. □

**Exercise 7.** What is wrong with the function `money` in Example 6? Why is it difficult for a C function to return a string (i.e., `char *`)? □

**Example 7: Factoring.** In algebra, we are accustomed to factoring expressions. For example, when we see the expression $Ax + Bx$ we are strongly tempted to write it in the form $(A + B)x$. Sometimes this is possible in PLs: we can certainly rewrite

```
z = A * x + B * x;
```

as

```
z = (A + B) * x;
```

Most PLs allow us to write something like this:

```
if (x > PI) y = sin(x) else y = cos(x)
```

Only a few PLs allow the shorter form:

```
y = if (x > PI) then sin(x) else cos(x);
```

Very few PLs recognize the even shorter form:

```
y = (if (x > PI) then sin else cos)(x);
```

□

**Example 8: Returning Functions.** Some PLs, such as Pascal, allow nested functions, whereas others, such as C, do not. Suppose that we were allowed to nest functions in C, and we defined the function shown in Listing 1.

The idea is that `addk` returns a function that "adds `k` to its argument". We would use it as in the following program, which should print "10":

```
int add6 (int) = addk(6);
printf("%d", add6(4));
```

□

**Exercise 8.** "It would be fairly easy to change C so that a function could return a function." True or false? (Note that you can represent the function itself by a pointer to its first instruction.) □

**Listing 1:** Returning a function

```
typedef int intint(int);
intint addk (int k)
{
  int f (int n)
  {
    return n + k;
  }
  return f;
}
```

**Example 9: Functions as Values.** Example 8 is just one of several oddities about the way programs handle functions. Some PLs do not permit this sequence:

```
if (x > PI) f = sin else f = cos;
f(x);
```

□

**Exercise 9.** Why are statements like this permitted in C (with appropriate syntax) but not in Pascal? □

**Example 10: Functions that work in more than one way.** The function `Add(x,y,z)` attempts to satisfy the constraint $x + y = z$. For instance:

- ▷ `Add(2,3,n)` assigns 5 to $n$.
- ▷ `Add(i,3,5)` assigns 2 to $i$.

□

**Example 11: Logic computation.** It would be convenient if we could encode logical assertions such as

$$\forall i \,.\, 0 < i < N \Rightarrow a[i-1] < a[i]$$

in the form

```
all (int i = 1; i < N; i++)
  a[i-a] < a[i]
```

and

$$\exists i \,.\, 0 \le i < N \wedge a[i] = k$$

in the form

```
exists (int i = 0; i < N; i++)
  a[i] == k
```

□

**Example 12: Implicit Looping.** It is a well-known fact that we can rewrite programs that use loops as programs that use recursive functions. It is less well-known that we can eliminate the recursion as well: see Listing 2.

**Listing 2:** The Factorial Function

```
        typedef TYPE ....
        TYPE fac (int n, TYPE f)
        {
          if (n <= 1)
            return 1;
          else
            return n * f(n-1, f);
        }
        printf("%d", fac(3, fac));
```

This program computes factorials using neither loops nor recursion:

$$\begin{aligned} \texttt{fac}(3, \texttt{fac}) &= 3 \times \texttt{fac}(2, \texttt{fac}) \\ &= 3 \times 2 \times \texttt{fac}(1, \texttt{fac}) \\ &= 3 \times 2 \times 1 \\ &= 6 \end{aligned}$$

In early versions of Pascal, functions like this actually worked. When type-checking was improved, however, they would no longer compile. □

**Exercise 10.** Can you complete the definition of `TYPE` in Example 2? □

The point of these examples is to show that PLs affect the way we think. It would not occur to most programmers to write programs in the style of the preceding sections because most PLs do not permit these constructions. Working with low-level PLs leads to the belief that "this is all that computers can do".

Garbage collection provides an example. Many programmers have struggled for years writing programs that require explicit deallocation of memory. Memory "leaks" are one of the most common forms of error in large programs and they can do serious damage. (One of the Apollo missions almost failed because of a memory leak that was corrected shortly before memory was exhausted, when the rocket was already in lunar orbit.) These programmers may be unaware of the existence of PLs that provide garbage collection, or they may believe (because they have been told) that garbage collection has an unacceptably high overhead.

# 3 Theoretical Issues

As stated in Section 1, theory is not a major concern of the course. In this section, however, we briefly review the contributions that theory has made to the development of PLs and discuss the importance and relevance of these contributions.

## 3.1 Syntactic and Lexical Issues

There is a large body of knowledge concerning formal languages. There is a hierarchy of languages (the "Chomsky hierarchy") and, associated with each level of the hierarchy, formal machines that can "recognize" the corresponding languages and perform other tasks. The first two levels of the hierarchy are the most familiar to programmers: regular languages and finite state automata; and context-free languages and push-down automata. Typical PLs have context-free (or almost context-free) grammars, and their tokens, or lexemes, can be described by a regular language. Consequently, programs can be split into lexemes by a finite state automaton and parsed by a push-down automaton.

It might appear, then, that the problem of PL design at this level is "solved". Unfortunately, there are exceptions. The grammar of C++, for example, is not context-free. This makes the construction of a C++ parser unnecessarily difficult and accounts for the slow development of C++ programming tools. The preprocessor is another obstacle that affects program development environments in both C and C++.

**Exercise 11.** Give examples to demonstrate the difficulties that the C++ preprocessor causes in a program development environment. □

## 3.2 Semantics

There are many kinds of semantics, including: algebraic, axiomatic, denotational, operational, and others. They are not competitive but rather complementary. There are many applications of semantics, and a particular semantic system may be more or less appropriate for a particular task.

For example, a denotational semantics is a mapping from program constructs to abstract mathematical entities that represent the "meaning" of the constructs. Denotational semantics is an effective language for communication between the designer and implementors of a PL but is not usually of great interest to a programmer who uses the PL. An axiomatic semantics, on the other hand, is not very useful to an implementor but may be very useful to a programmer who wishes to prove the correctness of a program.

The basic ideas of denotational semantics are due to Landin (1965) and Strachey (1966). The most complete description was completed by Milne (1976) after Strachey's death. Denotational semantics is a powerful descriptive tool: it provides techniques for mapping almost any PL feature into a high order mathematical function. Denotational semantics developed as a descriptive technique and was extended to handle increasingly arcane features, such as jumps into blocks.

Ashcroft and Wadge published an interesting paper (1982), noting that PL features that were easy to implement were often hard to describe and *vice versa.* They suggested that denotational semantics should be used ***prescriptively*** rather then ***descriptively***. In other words, a PL designer should start with a simple denotational semantics and then figure out how to implement the language — or perhaps leave that task to implementors altogether. Their own language, Lucid, was designed according to these principles.

To some extent, the suggestions of Ashcroft and Wadge have been followed (although not necessarily as a result of their paper). A description of a new PL in the academic literature is usually accompanied by a denotational semantics for the main features of the PL. It is not clear, however, that semantic techniques have had a significant impact on mainstream language design.

## 3.3 Type Theory

There is a large body of knowledge on type theory. We can summarize it concisely as follows. Here is a function declaration.

$T\ f(S\ x)\ \{\ B;\ \texttt{return}\ y;\ \}$

The declaration introduces a function called $f$ that takes an argument, $x$ of type $S$, performs calculations indicated as $B$, and returns a result, $y$ of type $T$. If there is a type theory associated with the language, we should be able to prove a theorem of the following form:

> If $x$ has type $S$ then the evaluation of $f(x)$ yields a value of type $T$.

The reasoning used in the proof is based on the syntactic form of the function body, $B$.

If we can do this for all legal programs in a language $\mathcal{L}$, we say that $\mathcal{L}$ is ***statically typed***. If $\mathcal{L}$ is indeed statically typed:

- ▷ A compiler for $\mathcal{L}$ can check the type correctness of all programs.
- ▷ A program that is type-correct will not fail because of a type error when it is executed.

A program that is type-correct may nevertheless fail in various ways. Type checking does not usually detect errors such as division by zero. (In general, most type checking systems assume that functions are ***total***: a function with type $S \to T$, given a value of type $S$, will return a value of type $T$. Some commonly used functions are ***partial***: for example $x/y \equiv$ divide(x,y) is undefined for $y = 0$.) A program can be type-correct and yet give completely incorrect answers. In general:

- ▷ Most PLs are not statically typed in the sense defined above, although the number of loopholes in the type system may be small.
- ▷ A type-correct program may give incorrect answers.
- ▷ Type theories for modern PLs, with objects, classes, inheritance, overloading, genericity, dynamic binding, and other features, are very complex.

Type theory leads to attractive and interesting mathematics and, consequently, tends to be overvalued by the theoretical Computer Science community. The theory of program correctness is more difficult and has attracted less attention, although it is arguably more important.

**Exercise 12.** Give an example of an expression or statement in Pascal or C that contains a type error that the compiler cannot detect. □

## 3.4 Regular Languages

Regular languages are based on a simple set of operations: sequence, choice, and repetition. Since these operations occur in many contexts in the study of PLs, it is interesting to look briefly at regular languages and to see how the concepts can be applied.

In the formal study of regular languages, we begin with an alphabet, $\Sigma$, which is a finite set of symbols. For example, we might have $\Sigma = \{0, 1\}$. The set of all finite strings, including the empty string, constructed from the symbols of $\Sigma$ is written $\Sigma^*$.

We then introduce a variety of ***regular expressions***, which we will refer to as RE here. (The abbreviation RE is also used to stand for "recursively enumerable", but in this section it stands for "regular expression".) Each form of RE is defined by the set of strings in $\Sigma^*$ that it generates. This set is called a "regular language".

1. $\varnothing$ is RE and denotes the empty set.
2. $\epsilon$ (the string containing no symbols) is RE and denotes the set $\{\epsilon\}$.
3. For each symbol $x \in \Sigma$, $x$ is RE and denotes the set $\{x\}$.
4. If $r$ is RE with language $R$, then $r^*$ is RE and denotes the set $R^*$ (where
$$R^* \quad = \quad \{ x_1 x_2 \ldots x_n \mid n \geq 0 \wedge x_i \in R \} \text{ ).}$$
5. If $r$ and $s$ are RE with languages $R$ and $S$ respectively, then $(r + s)$ is RE and denotes the set $R \cup S$.
6. If $r$ and $s$ are RE with languages $R$ and $S$ respectively, then $(rs)$ is RE and denotes the set $RS$ (where
$$RS \quad = \quad \{ rs \mid r \in R \wedge s \in S \} \text{ ).}$$

We add two further notations that serve solely as abbreviations.

1. The expression $a^n$ represents the string $aa \ldots a$ containing $n$ occurrences of the symbol $a$.
2. The expression $a^+$ is an abbreviation for the RE $aa^*$ that denotes the set $\{a, aa, aaa, \ldots\}$.

### 3.4.1 Tokens

We can use regular languages to describe the tokens (or lexemes) of most PLs. For example:

$$\begin{array}{rcl}
\texttt{UC} & = & A + B + \cdots + Z \\
\texttt{LC} & = & a + b + \cdots + z \\
\texttt{LETTER} & = & UC + LC \\
\texttt{DIGIT} & = & 0 + 1 + \cdots + 9 \\
\texttt{IDENTIFIER} & = & \texttt{LETTER}(\ \texttt{LETTER} + \texttt{DIGIT})^* \\
\texttt{INTCONST} & = & \texttt{DIGIT}^+ \\
\texttt{FLOATCONST} & = & \texttt{DIGIT}^+\ \texttt{.}\ \texttt{DIGIT}^* \\
 & \ldots . & 
\end{array}$$

We can use a program such as `lex` to construct a lexical analyzer (scanner) from a regular expression that defines the tokens of a PL.

**Exercise 13.** Some compilers allow "nested comments". For example, $\{\ldots\{\ldots\}\ldots\}$ in Pascal or $/*\ldots/*\ldots*/\ldots*/$ in C. What consequences does this have for the lexical analyzer (scanner)? □

| RE | Control Structure |
|---|---|
| $x$ | statement |
| $rs$ | statement; statement |
| $r^*$ | `while` expression `do` statement |
| $r+s$ | `if` condition `then` statement `else` statement |

Figure 1: REs and Control Structures

### 3.4.2 Context Free Grammars

Context free grammars for PLs are usually written in BNF (Backus-Naur Form). A grammar rule, or ***production***, consists of a non-terminal symbol (the symbol being defined), a connector (usually ::= or →), and a sequence of terminal and non-terminal symbols. The syntax for the assignment, for example, might be defined:

$$\texttt{ASSIGNMENT} \quad \rightarrow \quad \texttt{VARIABLE} \text{ `` := '' } \texttt{EXPRESSION}$$

Basic BNF provides no mechanism for choice: we must provide one rule for each possibility. Also, BNF provides no mechanism for repetition; we must use recursion instead. The following example illustrates both of these limitations.

$$\begin{aligned} \texttt{SEQUENCE} &\quad \rightarrow \quad \texttt{EMPTY} \\ \texttt{SEQUENCE} &\quad \rightarrow \quad \texttt{STATEMENT}\;\texttt{SEQUENCE} \end{aligned}$$

BNF has been extended in a variety of ways. With a few exceptions, most of the extended forms can be described simply: the sequence of symbols on the right of the connector is replaced by a regular expression. The extension enables us to express choice and repetition within a single rule. In grammars, the symbol | rather than + is used to denote choice.

$$\begin{aligned} \texttt{STATEMENT} &= \texttt{ASSIGNMENT} \mid \texttt{CONDITIONAL} \mid \cdots \\ \texttt{SEQUENCE} &= (\,\texttt{STATEMENT}\,)^* \end{aligned}$$

Extended BNF (EBNF) provides a more concise way of describing grammars than BNF. Just as parsers can be constructed from BNF grammars, they can be constructed from EBNF grammars.

**Exercise 14.** Discuss the difficulty of adding attributes to an EBNF grammar. □

### 3.4.3 Control Structures and Data Structures

Figure 1 shows a correspondence between REs and the control structures of Algol-like languages. Similarly, Figure 2 shows a correspondence between REs and data structures..

**Exercise 15.** Why might Figures 1 and 2 be of interest to a PL designer? □

| RE | Data Structure |
|---|---|
| $x$ | `int` $n$; |
| $rs$ | `struct` { `int` $n$; `float` $y$; } |
| $r^n$ | `int` $a[n]$; |
| $r^*$ | `int` $a[]$ |
| $r+s$ | `union` { `int` $n$; `float` $y$; } |

Figure 2: REs and Data Structures

### 3.4.4 Discussion

These analogies suggest that the mechanisms for constructing REs — concatenation, alternation, and repetition — are somehow fundamental. In particular, the relation between the standard control structures and the standard data structures can be helpful in programming. In Jackson Structured Design (JSD), for example, the data structures appropriate for the application are selected first and then the program is constructed with the corresponding control structures.

The limitations of REs are also interesting. When we move to the next level of the Chomsky hierarchy, Context Free Languages (CFLs), we obtain the benefits of recursion. The corresponding control structure is the ***procedure*** and the corresponding data structures are recursive structures such as lists and trees (Wirth 1976, page 163).

# 4 The Procedural Paradigm

The introduction of the von Neumann architecture was a crucial step in the development of electronic computers. The basic idea is that instructions can be encoded as data and stored in the memory of the computer. The first consequence of this idea is that changing and modifying the stored program is simple and efficient. In fact, changes can take place at electronic speeds, a very different situation from earlier computers that were programmed by plugging wires into panels. The second, and ultimately more far-reaching, consequence is that computers can process programs themselves, under program control. In particular, a computer can translate a program from one notation to another. Thus the stored-program concept led to the development of programming "languages".

The history of PLs, like the history of any technology, is complex. There are advances and setbacks; ideas that enter the mainstream and ideas that end up in a backwater; even ideas that are submerged for a while and later surface in an unexpected place.

With the benefit of hindsight, we can identify several strands in the evolution of PLs. These strands are commonly called "paradigms" and, in this course, we survey the paradigms separately although their development was interleaved.

Sources for this section include (Wexelblat 1981; Williams and Campbell-Kelly 1989; Bergin and Gibson 1996).

## 4.1 Early Days

The first PLs evolved from machine code. The first programs used numbers to refer to machine addresses. One of the first additions to programming notation was the use of symbolic names rather than numbers to represent addresses.

> Briefly, it enables the programmer to refer to any word in a programme by means of a label or tag attached to it arbitrarily by the programmer, instead of by its address in the store. Thus, for example, a number appearing in the calculation might be labelled '`a3`'. The programmer could then write simply '`A a3`' to denote the operation of adding this number into the accumulator, without having to specify just where the number is located in the machine. (Mutch and Gill 1954)

The key point in this quotation is the phrase "instead of by its address in the store". Instead of writing

| Location | Order |
|---|---|
| 100 | A 104 |
| 101 | A 2 |
| 102 | T 104 |
| 103 | H 24 |
| 104 | C 50 |
| 105 | T 104 |

the programmer would write

```
        A   a3
        A   2
        T   a3
        H   24
   a3)  C   50
        T   a3
```

systematically replacing the ***address*** `104` by the ***symbol*** `a3`. This establishes the principle that a variable name stands for a memory location, a principle that influenced the subsequent development of PLs and is now known — perhaps inappropriately — as ***value semantics***.

The importance and subroutines and subroutine libraries was recognized before high-level programming languages had been developed, as the following quotation shows.

> The following advantages arise from the use of such a library:
>
> 1. It simplifies the task of preparing problems for the machine;
> 2. It enables routines to be more readily understood by other users, as conventions are standardized and the units of a routine are much larger, being subroutines instead of individual orders;
> 3. Library subroutines may be used directly, without detailed coding and punching;
> 4. Library subroutines are known to be correct, thus greatly reducing the overall chance of error in a complete routine, and making it much easier to locate errors.
>
> . . . . Another difficulty arises from the fact that, although it is desirable to have subroutines available to cover all possible requirements, it is also undesirable to allow the size of the resulting library to increase unduly. However, a subroutine can be made more versatile by the use of parameters associated with it, thus reducing the total size of the library.
>
> We may divide the parameters associated with subroutines into two classes.
>
> EXTERNAL parameters, i.e. parameters which are fixed throughout the solution of a problem and arise solely from the use of the library;
>
> INTERNAL parameters, i.e. parameters which vary during the solution of the problem.
>
> . . . . Subroutines may be divided into two types, which we have called OPEN and CLOSED. An open subroutine is one which is included in the routine as it stands whereas a closed subroutine is placed in an arbitrary position in the store and can be called into use by any part of the main routine. (Wheeler 1951)

**Exercise 16.** This quotation introduces a theme that has continued with variations to the present day. Find in it the origins of concern for:

- ▷ correctness;
- ▷ maintenance;
- ▷ encapsulation;
- ▷ parameterization;
- ▷ genericity;
- ▷ reuse;

▷ space/time trade-offs.

□

Machine code is a sequence of "orders" or "instructions" that the computer is expected to execute. The style of programming that this viewpoint developed became known as the "imperative" or "procedural" programming paradigm. In these notes, we use the term "procedural" rather than "imperative" because programs resemble "procedures" (in the English, non-technical sense) or recipes rather than "commands". Confusingly, the individual steps of procedural PLs, such as Pascal and C, are often called "statements", although in logic a "statement" is a sentence that is either true or false.

By default, the commands of a procedural program are executed sequentially. Procedural PLs provide various ways of escaping from the sequence. The earliest mechanisms were the "jump" command, which transferred control to another part of the program, and the "jump and store link" command, which transferred control but also stored a "link" to which control would be returned after executing a subroutine.

The data structures of these early languages were usually rather simple: typically primitive values (integers and floats) were provided, along with single- and multi-dimensioned arrays of primitive values.

## 4.2 FORTRAN

FORTRAN was introduced in 1957 at IBM by a team led by John Backus. The "Preliminary Report" describes the goal of the FORTRAN project:

> The IBM Mathematical Formula Translation System or briefly, FORTRAN, will comprise a large set of programs to enable the IBM 704 to accept a concise formulation of a problem in terms of a mathematical notation and to produce automatically a high-speed 704 program for the solution of the problem. (Quoted in (Sammet 1969).)

This suggests that the IBM team's goal was to eliminate programming! The following quotation seems to confirm this:

> If it were possible for the 704 to code problems for itself and produce as good programs as human coders (but without the errors), it was clear that large benefits could be achieved. (Backus 1957)

It is interesting to note that, 20 years later, Backus (1978) criticized FORTRAN and similar languages as "lacking useful mathematical properties". He saw the assignment statement as a source of inefficiency: "the von Neumann bottleneck". The solution, however, was very similar to the solution he advocated in 1957 — programming must become more like mathematics: "we should be focusing on the form and content of the overall result".

Although FORTRAN did not eliminate programming, it was a major step towards the elimination of assembly language coding. The designers focused on efficient implementation rather than elegant language design, knowing that acceptance depended on the high performance of compiled programs.

FORTRAN has value semantics. Variable names stand for memory addresses that are determined when the program is loaded.

The major achievements of FORTRAN are:

- ▷ efficient compilation;
- ▷ separate compilation (programs can be presented to the compiler as separate subroutines, but the compiler does not check for consistency between components);
- ▷ demonstration that high-level programming, with automatic translation to machine code, is feasible.

The principal limitations of FORTRAN are:

**Flat, uniform structure.** There is no concept of nesting in FORTRAN. A program consists of a sequence of subroutines and a main program. Variables are either global or local to subroutines. In other words, FORTRAN programs are rather similar to assembly language programs: the main difference is that a typical line of FORTRAN describes evaluating an expression and storing its value in memory whereas a typical line of assembly language specifies a machine instruction (or a small group of instructions in the case of a macro).

**Limited control structures.** The control structures of FORTRAN are `IF`, `DO`, and `GOTO`. Since there are no compound statements, labels provide the only indication that a sequence of statements form a group.

**Unsafe memory allocation.** FORTRAN borrows the concept of `COMMON` storage from assembly language program. This enables different parts of a program to share regions of memory, but the compiler does not check for consistent usage of these regions. One program component might use a region of memory to store an array of integers, and another might assume that the same region contains reals. To conserve precious memory, FORTRAN also provides the `EQUIVALENCE` statement, which allows variables with different names and types to share a region of memory.

**No recursion.** FORTRAN allocates all data, including the parameters and local variables of subroutines, statically. Recursion is forbidden because only one instance of a subroutine can be active at one time.

**Exercise 17.** The FORTRAN 1966 Standard stated that a FORTRAN implementation ***may*** allow recursion but is not required to do so. How would you interpret this statement if you were:

- ▷ writing a FORTRAN program?
- ▷ writing a FORTRAN compiler?

□

## 4.3 Algol 60

During the late fifties, most of the development of PLs was coming from industry. IBM dominated, with COBOL, FORTRAN, and FLPL (FORTRAN List Processing Language), all designed for the IBM 704. Algol 60 (Naur et al. 1960; Naur 1978; Perlis 1978) was designed by an international committee, partly to provide a PL that was independent of any particular company and its computers. The committee included both John Backus (chief designer of FORTRAN) and John McCarthy (designer of LISP).

The goal was a "universal programming language". In one sense, Algol was a failure: few complete, high-quality compilers were written and the language was not widely used (although it was used

**Listing 3:** An Algol Block

```
begin
  integer x;
  begin
    integer x;
    real y;
    x := 2;
    y := 3.14159;
  end;
  x := 1;
end
```

more in Europe than in North America). In another sense, Algol was a huge success: it became the standard language for describing algorithms. For the better part of 30 years, the ACM required submissions to the algorithm collection to be written in Algol.

The major innovations of Algol are discussed below.

**Block Structure.** Algol programs are recursively structured. A program is a ***block***. A block consists of declarations and statements. There are various kinds of statement; in particular, one kind of statement is a block. A variable or function name declared in a block can be accessed only within the block: thus Algol introduced ***nested scopes***. The recursive structure of programs means that large programs can be constructed from small programs. In the Algol block shown in Listing 3, the two assignments to `x` refer to two different variables.

The run-time entity corresponding to a block is called an ***activation record*** (AR). The AR is created on entry to the block and destroyed after the statements of the block have been executed. The syntax of Algol ensures that blocks are fully nested; this in turn means that ARs can be allocated on a ***stack***. Block structure and stacked ARs have been incorporated into almost every language since Algol.

**Dynamic Arrays.** The designers of Algol realized that it was relatively simple to allow the size of an array to be determined at run-time. The compiler statically allocates space for a pointer and an integer (collectively called a "dope vector") on the stack. At run-time, when the size of the array is known, the appropriate amount of space is allocated on the stack and the components of the "dope vector" are initialized. The following code works fine in Algol 60.

```
procedure average (n); integer n;
  begin
    real array a[1:n];
    ....
  end;
```

Despite the simplicity of the implementation, successor PLs such as C and Pascal dropped this useful feature.

**Call By Name.** The default method of passing parameters in Algol was "call by name" and it was described by a rather complicated "copy rule". The essence of the copy rule is that the program behaves ***as if*** the text of the formal parameter in the function is replaced by the text of the actual parameter. The complications arise because it may be necessary to rename some of the variables during the textual substitution. The usual implementation strategy

**Listing 4:** Call by name

```
procedure count (n); integer n;
  begin
    n := n + 1
  end
```

**Listing 5:** A General Sum Function

```
integer procedure sum (max, i, val); integer max, i, val;
  begin
    integer s;
    s := 0;
    for i := 1 until n do
      s := s + val;
    sum := s
  end
```

was to translate the actual parameter into a procedure with no arguments (called a "thunk"); each occurrence of the formal parameter inside the function was translated into a call to this function.

The mechanism seems strange today because few modern languages use it. However, the Algol committee had several valid reasons for introducing it.

- ▷ Call by name enables procedures to alter their actual parameters. If the procedure `count` is defined as in Listing 4, the statement

  ```
  count(widgets)
  ```

  has the same effect as the statement

  ```
  begin
    widgets := widgets + 1
  end
  ```

  The other parameter passing mechanism provided by Algol, call by value, does not allow a procedure to alter the value of its actual parameters in the calling environment: the parameter behaves like an initialized local variable.

- ▷ Call by name provides control structure abstraction. The procedure in Listing 5 provides a form of abstraction of a `for` loop. The first parameter specifies the number of iterations, the second is the loop index, and the third is the loop body. The statement

  ```
  sum(3, i, a[i])
  ```

  computes `a[1]+a[2]+a[3]`.

- ▷ Call by name evaluates the actual parameter exactly as often as it is accessed. (This is in contrast with call by value, where the parameter is usually evaluated exactly once, on entry to the procedure.) For example, if we declare the procedure `try` as in Listing 6, it is safe to call `try(x > 0, 1.0/x)`, because, if $x \leq 0$, the expression $1.0/x$ will not be evaluated.

**Own Variables.** A variable in an Algol procedure can be declared `own`. The effect is that the variable has local ***scope*** (it can be accessed only by the statements within the procedure) but global ***extent*** (its lifetime is the execution of the entire program).

**Listing 6:** Using call by name

```
real procedure try (b, x); boolean b; real x;
  begin
    try := if b then x else 0.0
  end
```

**Exercise 18.** Why does `i` appear in the parameter list of `Sum`? □

**Exercise 19.** Discuss the initialization of `own` variables. □

Algol 60 and most of its successors, like FORTRAN, has a value semantics. A variable name stands for a memory addresses that is determined when the block containing the variable declaration is entered at run-time.

With hindsight, we can see that Algol made important contributions but also missed some very interesting opportunities.

- ▷ An Algol block without statements is, in effect, a record. Yet Algol 60 does not provide records.
- ▷ The local data of an AR is destroyed after the statements of the AR have been executed. If the data was retained rather than destroyed, Algol would be a language with `modules`.
- ▷ An Algol block consists of declarations followed by statements. Suppose that declarations and statements could be interleaved in a block. In the following block, $D$ denotes a sequence of declarations and $S$ denotes a sequence of statements.

  **begin**<br>
  $D_1$<br>
  $S_1$<br>
  $D_2$<br>
  $S_2$<br>
  **end**

  A natural interpretation would be that $S_1$ and $S_2$ are executed ***concurrently***.
- ▷ `Own` variables were in fact rather problematic in Algol, for various reasons including the difficulty of reliably initializing them (see Example 19). But the concept was important: it is the separation of scope and extent that ultimately leads to objects.
- ▷ The call by name mechanism was a first step towards the important idea that functions can be treated as values. The actual parameter in an Algol call, assuming the default calling mechanism, is actually a parameterless procedure, as mentioned above. Applying this idea consistently throughout the language would have led to high order functions and paved the way to functional programming.

The Algol committee knew what they were doing, however. They knew that incorporating the "missed opportunities" described above would have led to significant implementation problems. In particular, since they believed that the stack discipline obtained with nested blocks was crucial for efficiency, anything that jeopardized it was not acceptable.

Algol 60 was simple and powerful, but not quite powerful enough. The dominant trend after Algol was towards languages of increasing complexity, such as PL/I and Algol 68. Before discussing these, we take a brief look at COBOL.

## 4.4 COBOL

COBOL (Sammett 1978) introduced structured data and implicit type conversion. When COBOL was introduced, "programming" was more or less synonymous with "numerical computation". COBOL introduced "data processing", where data meant large numbers of characters. The data division of a COBOL program contained descriptions of the data to be processed.

Another important innovation of COBOL was a new approach to data types. The problem of type conversion had not arisen previously because only a small number of types were provided by the PL. COBOL introduced many new types, in the sense that data could have various degrees of precision, and different representations as text. The choice made by the designers of COBOL was radical: type conversion should be automatic.

The assignment statement in COBOL has several forms, including

```
MOVE X to Y.
```

If X and Y have different types, the COBOL compiler will attempt to find a conversion from one type to the other. In most PLs of the time, a single statement translated into a small number of machine instructions. In COBOL, a single statement could generate a large amount of machine code.

**Example 13: Automatic conversion in COBOL.** The Data Division of a COBOL program might contain these declarations:

```
77    SALARY PICTURE 99999, USAGE IS COMPUTATIONAL.
77    SALREP PICTURE $$$,$$9.99
```

The first indicates that SALARY is to be stored in a form suitable for computation (probably, but not necessarily, binary form) and the second provides a format for reporting salaries as amounts in dollars. (Only one dollar symbol will be printed, immediately before the first significant digit). The Procedure Division would probably contain a statement like

```
MOVE SALARY TO SALREP.
```

which implicitly requires the conversion from binary to character form, with appropriate formatting. □

**Exercise 20.** Despite significant advances in the design and implementation of PLs, it remains true that FORTRAN is widely used for "number crunching" and COBOL is widely used for data processing. Can you explain why? □

## 4.5 PL/I

During the early 60s, the dominant languages were Algol, COBOL, FORTRAN. The continuing desire for a "universal language" that would be applicable to a wide variety of problem domains led IBM to propose a new programming language (originally called NPL but changed, after objections from the UK's National Physical Laboratory, to PL/I) that would combine the best features of these three languages. Insiders at the time referred to the new language as "CobAlgoltran".

The design principles of PL/I (Radin 1978) included:

- ▷ the language should contain the features necessary for all kinds of programming;
- ▷ a programmer could learn a subset of the language, suitable for a particular application, without having to learn the entire language.

An important lesson of PL/I is that these design goals are doomed to failure. A programmer who has learned a "subset" of PL/I is likely, like all programmers, to make a mistake. With luck, the compiler will detect the error and provide a diagnostic message that is incomprehensible to the programmer because it refers to a part of the language outside the learned subset. More probably, the compiler will *not* detect the error and the program will behave in a way that is inexplicable to the programmer, again because it is outside the learned subset.

PL/I extends the automatic type conversion facilities of COBOL to an extreme degree. For example, the expression (Gelernter and Jagannathan 1990)

```
('57' || 8) + 17
```

is evaluated as follows:

1. Convert the integer 8 to the string `'8'`.
2. Concatenate the strings `'57'` and `'8'`, obtaining `'578'`.
3. Convert the string `'578'` to the integer 578.
4. Add 17 to 578, obtaining 595.
5. Convert the integer 595 to the string `'595'`.

The compiler's policy, on encountering an assignment $x = E$, might be paraphrased as: "Do everything possible to compile this statement; as far as possible, avoid issuing any diagnostic message that would tell the programmer what is happening".

PL/I did introduce some important new features into PLs. They were not all well-designed, but their existence encouraged others to produce better designs.

- ▷ Every variable has a ***storage class***: `static`, `automatic`, `based`, or `controlled`. Some of these were later incorporated into C.
- ▷ An object associated with a `based` variable $x$ requires explicit allocation and is placed on the heap rather than the stack. Since we can execute the statement `allocate` $x$ as often as necessary, `based` variables provide a form of template.
- ▷ PL/I provides a wide range of programmer-defined types. Types, however, could not be named.
- ▷ PL/I provided a simple, and not very safe, form of exception handling. Statements of the following form are allowed anywhere in the program:

```
ON condition
  BEGIN;
  ....
  END;
```

  If the `condition` (which might be `OVERFLOW`, `PRINTER OUT OF PAPER`, etc.) becomes `TRUE`, control is transferred to whichever `ON` statement for that condition was most recently executed. After the statements between `BEGIN` and `END` (the handler) have been executed, control returns to the statement that raised the exception or, if the handler contains a `GOTO` statement, to the target of that statement.

**Exercise 21.** Discuss potential problems of the PL/I exception handling mechanism. □

## 4.6 Algol 68

Whereas Algol 60 is a simple and expressive language, its successor Algol 68 (van Wijngaarden et al. 1975; Lindsey 1996) is much more complex. The main design principle of Algol 68 was ***orthogonality***: the language was to be defined using a number of basic concepts that could be combined in arbitrary ways. Although it is true that *lack of* orthogonality can be a nuisance in PLs, it does not necessarily follow that orthogonality is always a good thing.

The important features introduced by Algol 68 include the following.

- ▷ The language was described in a formal notation that specified the complete syntax and semantics of the language (van Wijngaarden et al. 1975). The fact that the Report was very hard to understand may have contributed to the slow acceptance of the language.
- ▷ Operator overloading: programmers can provide new definitions for standard operators such as "+". Even the priority of these operators can be altered.
- ▷ Algol 68 has a very uniform notation for declarations and other entities. For example, Algol 68 uses the same syntax (`mode name = expression`) for types, constants, variables, and functions. This implies that, for all these entities, there must be forms of `expression` that yield appropriate values.
- ▷ In a ***collateral clause*** of the form $(x, y, z)$, the expressions $x$, $y$, and $z$ can be evaluated in any order, or concurrently. In a function call $f(x, y, z)$, the argument list is a collateral clause.

  Collateral clauses provide a good, and early, example of the idea that a PL specification should intentionally leave some implementation details undefined. In this example, the Algol 68 report does not specify the order of evaluation of the expressions in a collateral clause. This gives the implementor freedom to use any order of evaluation and hence, perhaps, to optimize.
- ▷ The operator `ref` stands for "reference" and means, roughly, "use the address rather than the value". This single keyword introduces call by reference, pointers, dynamic data structures, and other features to the language. It appears in C in the form of the operators "`*`" and "`&`".
- ▷ A large vocabulary of PL terms, some of which have become part of the culture (cast, coercion, narrowing, . . . .) and some of which have not (mode, weak context, voiding,. . . .).

Like Algol 60, Algol 68 was not widely used, although it was popular for a while in various parts of Europe. The ideas that Algol 68 introduced, however, have been widely imitated.

**Exercise 22.** Algol 68 has a rule that requires, for an assignment $x := E$, the lifetime of the variable $x$ must be less than or equal to the lifetime of the object obtained by evaluating $E$. Explain the motivation for this rule. To what extent can it be checked by the compiler? □

## 4.7 Pascal

Pascal was designed by Wirth (1996) as a reaction to the complexity of Algol 68, PL/I, and other languages that were becoming popular in the late 60s. Wirth made extensive use of the ideas of Dijkstra and Hoare (later published as (Dahl, Dijkstra, and Hoare 1972)), especially Hoare's ideas of data structuring. The important contributions of Pascal included the following.

- ▷ Pascal demonstrated that a PL could be simple yet powerful.
- ▷ The type system of Pascal was based on primitives (`integer`, `real`, `bool`, . . . .) and mechanisms for building structured types (`array`, `record`, `file`, `set`, . . . .). Thus data types in Pascal form a recursive hierarchy just as blocks do in Algol 60.

- ▷ Pascal provides no implicit type conversions other than subrange to `integer` and `integer` to `real`. All other type conversions are explicit (even when no action is required) and the compiler checks type correctness.
- ▷ Pascal was designed to match Wirth's (1971) ideas of program development by stepwise refinement. Pascal is a kind of "fill in the blanks" language in which all programs have a similar structure, determined by the relatively strict syntax. Programmers are expected to start with a complete but skeletal "program" and flesh it out in a series of refinement steps, each of which makes certain decisions and adds new details. The monolithic structure that this idea imposes on programs is a drawback of Pascal because it prevents independent compilation of components.

Pascal was a failure because it was too simple. Because of the perceived missing features, supersets were developed and, inevitably, these became incompatible. The first version of "Standard Pascal" was almost useless as a practical programming language and the Revised Standard described a usable language but appeared only after most people had lost interest in Pascal.

Like Algol 60, Pascal missed important opportunities. The `record` type was a useful innovation (although very similar to the Algol 68 `struct`) but allowed data only. Allowing functions in a record declaration would have paved the way to modular and even object oriented programming.

Nevertheless, Pascal had a strong influence on many later languages. Its most important innovations were probably the combination of simplicity, data type declarations, and static type checking.

**Exercise 23.** List some of the "missing features" of Pascal. □

**Exercise 24.** It is well-known that the biggest loop-hole in Pascal's type structure was the variant record. How serious do you think this problem was? □

## 4.8 Modula–2

Wirth (1982) followed Pascal with Modula–2, which inherits Pascal's strengths and, to some extent, removes Pascal's weaknesses. The important contribution of Modula–2 was, of course, the introduction of modules. (Wirth's first design, Modula, was never completed. Modula–2 was the product of a sabbatical year in California, where Wirth worked with the designers of Mesa, another early modular language.)

A module in Modula–2 has an ***interface*** and an ***implementation***. The interface provides information about the use of the module to both the programmer and the compiler. The implementation contains the "secret" information about the module. This design has the unfortunate consequence that some information that should be secret must be put into the interface. For example, the compiler must know the size of the object in order to declare an instance of it. This implies that the size must be deducible from the interface which implies, in turn, that the interface must contain the representation of the object. (The same problem appears again in C++.)

Modula–2 provides a limited escape from this dilemma: a programmer can define an "opaque" type with a hidden representation. In this case, the interface contains only a pointer to the instance and the representation can be placed in the implementation module.

The important features of Modula–2 are:

- ▷ Modules with separated interface and implementation descriptions (based on Mesa).
- ▷ Coroutines.

## 4.9 C

C is a very pragmatic PL. Ritchie (Ritchie 1996) designed it for a particular task — systems programming — for which it has been widely used. The enormous success of C is partly accidental. UNIX, after Bell released it to universities, became popular, with good reason. Since UNIX depended heavily on C, the spread of UNIX inevitably led to the spread of C.

C is based on a small number of primitive concepts. For example, arrays are defined in terms of pointers and pointer arithmetic. This is both the strength and weakness of C. The number of concepts is small, but C does not provide real support for arrays, strings, or boolean operations.

C is a low-level language by comparison with the other PLs discussed in this section. It is designed to be easy to compile and to produce efficient object code. The compiler is assumed to be rather unsophisticated (a reasonable assumption for a compiler running on a PDP/11 in the late sixties) and in need of hints such as `register`. C is notable for its concise syntax. Some syntactic features are inherited from Algol 68 (for example, `+=` and other assignment operators) and others are unique to C and C++ (for example, postfix and prefix `++` and `--`).

## 4.10 Ada

Ada (Whitaker 1996) represents the last major effort in procedural language design. It is a large and complex language that combines then-known programming features with little attempt at consolidation. It was the first widely-used language to provide full support for concurrency, with interactions checked by the compiler, but this aspect of the language proved hard to implement.

Ada provides templates for procedures, record types, generic packages, and task types. The corresponding objects are: blocks and records (representable in the language); and packages and tasks (not representable in the language). It is not clear why four distinct mechanisms are required (Gelernter and Jagannathan 1990). The syntactic differences suggest that the designers did not look for similarities between these constructs. A procedure definition looks like this:

```
procedure procname ( parameters ) is
  body
```

A record type looks like this:

```
type recordtype ( parameters ) is
  body
```

The parameters of a record type are optional. If present, they have a different form than the parameters of procedures.

A generic package looks like this:

```
generic ( parameters ) package packagename is
  package description
```

The parameters can be types or values. For example, the template

```
generic
  max:  integer;
  type element is private;
package Stack is
....
```

might be instantiated by a declaration such as

```
        package intStack is new Stack(20, integer)
```

Finally, a task template looks like this (no parameters are allowed):

```
        task type templatename is
          task description
```

Of course, programmers hardly notice syntactic differences of this kind: they learn the correct incantation and recite it without thinking. But it is disturbing that the language ***designers*** apparently did not consider passible relationships between these four kinds of declaration. Changing the syntax would be a minor improvement, but uncovering deep semantic similarities might have a significant impact on the language as a whole, just as the identity declaration of Algol 68 suggested new and interesting possibilities.

**Exercise 25.** Propose a uniform style for Ada declarations. □

# 5 The Functional Paradigm

Procedural programming is based on instructions ("do something") but, inevitably, procedural PLs also provide expressions ("calculate something"). The key insight of functional programming (FP) is that everything can be done with expressions: the commands are unnecessary.

This point of view has a solid foundation in theory. Turing (1936) introduced an abstract model of "programming", now known as the Turing machine. Kleene (1936) and Church (1941) introduced the theory of recursive functions. The two theories were later shown (by Kleene) to be equivalent: each had the same computational power. Other theories, such as Post production systems, were shown to have the same power. This important theoretical result shows that FP is not a complete waste of time but it does not tell us whether FP is useful or practical. To decide that, we must look at the functional programming languages (FPLs) that have actually been implemented.

Most functional language support ***high order functions***. Roughly, a high order function is a function that takes another function as a parameter or returns a function. More precisely:

- ▷ A ***zeroth order*** expression contains only variables and constants.
- ▷ A ***first order*** expression may also contain function invocations, but the results and parameters of functions are variables and constants (that is, zeroth order expressions).
- ▷ In general, in an $n$***-th order*** expression, the results and parameters of functions are $n - 1$-th order expressions.
- ▷ A ***high order*** expression is an $n$-th order expression with $n \geq 2$.

The same conventions apply in logic with "function" replaced by "function or predicate". In first-order logic, quantifiers can bind variables only; in a high order logic, quantifiers can bind predicates.

## 5.1 LISP

Functional programming was introduced in 1958 in the form of LISP by John McCarthy. The following account of the development of LISP is based on McCarthy's (1978) history.

The important early decisions in the design of LISP were:

- ▷ to provide list processing (which already existed in languages such as Information Processing Language (IPL) and FORTRAN List Processing Language (FLPL));
- ▷ to use a prefix notation (emphasizing the operator rather than the operands of an expression);
- ▷ to use the concept of "function" as widely as possible (`cons` for list construction; `car` and `cdr` for extracting list components; `cond` for conditional, etc.);
- ▷ to provide higher order functions and hence a notation for functions (based on Church's (1941) $\lambda$-notation);
- ▷ to avoid the need for explicit erasure of unused list structures.

McCarthy (1960) wanted a language with a solid mathematical foundation and decided that recursive function theory was more appropriate for this purpose than the then-popular Turing machine model. He considered it important that LISP expressions should obey the usual mathematical laws allowing replacement of expressions and:

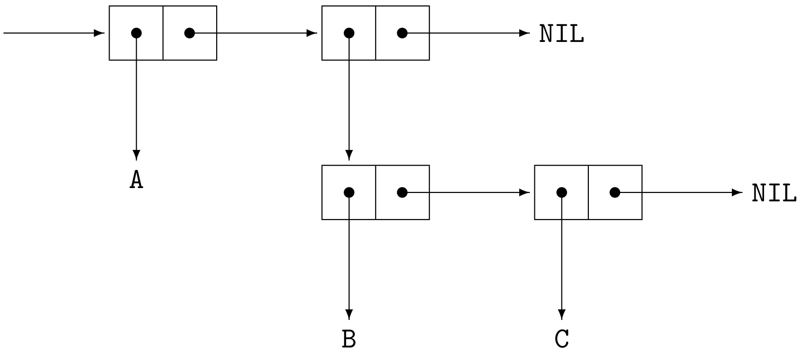

Figure 3: The list structure `(A (B C))`

> Another way to show that LISP was neater than Turing machines was to write a universal LISP function and show that it is briefer and more comprehensible than the description of a universal Turing machine. This was the LISP function *eval*$[e, a]$, which computes the value of a LISP expression $e$, the second argument $a$ being a list of assignments of values to variables. . . . Writing *eval* required inventing a notation for representing LISP functions as LISP data, and such a notation was devised for the purpose of the paper with no thought that it would be used to express LISP programs in practice. (McCarthy 1978)

After the paper was written, McCarthy's graduate student S. R. Russel noticed that *eval* could be used as an interpreter for LISP and hand-coded it, thereby producing the first LISP interpreter. Soon afterwards, Timothy Hart and Michael Levin wrote a LISP compiler in LISP; this is probably the first instance of a compiler written in the language that it compiled.

The function application $f(x, y)$ is written in LISP as `(f x y)`. The function name always comes first: $a + b$ is written in LISP as `(+ a b)`. All expressions are enclosed in parentheses and can be nested to arbitrary depth.

There is a simple relationship between the text of an expression and its representation in memory. An ***atom*** is a simple object such as a name or a number. A ***list*** is a data structure composed of `cons`-cells (so called because they are constructed by the function `cons`); each `cons`-cell has two pointers and each pointer points either to another `cons`-cell or to an atom. Figure 3 shows the list structure corresponding to the expression `(A (B C))`. Each box represents a `cons`-cell. There are two lists, each with two elements, and each terminated with `NIL`. The diagram is simplified in that the atoms `A`, `B`, `C`, and `NIL` would themselves be list structures in an actual LISP system.

The function `cons` constructs a list from its head and tail: `(cons head tail)`. The value of `(car list)` is the head of the list and the value of `(cdr list)` is the tail of the list. Thus:

$$\begin{aligned} \texttt{(car (cons head tail))} &\rightarrow \texttt{head} \\ \texttt{(cdr (cons head tail))} &\rightarrow \texttt{tail} \end{aligned}$$

The names `car` and `cdr` originated in IBM 704 hardware; they are abbreviations for "contents of address register" (the top 18 bits of a 36-bit word) and "contents of decrement register" (the bottom 18 bits).

It is easy to translate between list expressions and the corresponding data structures. There is a function `eval` (mentioned in the quotation above) that evaluates a stored list expression.

Consequently, it is straightforward to build languages and systems “on top of” LISP and LISP is often used in this way.

It is interesting to note that the close relationship between code and data in LISP mimics the von Neumann architecture at a higher level of abstraction.

LISP was the first in a long line of functional programming (FP) languages. Its principal contributions are listed below.

**Names.** In procedural PLs, a name denotes a storage location (value semantics). In LISP, a name is a reference to an object, not a location (reference semantics). In the Algol sequence

```
int n;
n := 2;
n := 3;
```

the declaration `int n;` assigns a name to a location, or “box”, that can contain an integer. The next two statements put different values, first 2 then 3, into that box. In the LISP sequence

```
(progn
  (setq x (car structure))
  (setq x (cdr structure)))
```

`x` becomes a reference first to `(car structure)` and then to `(cdr structure)`. The two objects have different memory addresses. A consequence of the use of names as references to objects is that eventually there will be objects for which there are no references: these objects are “garbage” and must be automatically reclaimed if the interpreter is not to run out of memory. The alternative — requiring the programmer to explicitly deallocate old cells — would add considerable complexity to the task of writing LISP programs. Nevertheless, the decision to include automatic garbage collection (in 1958!) was courageous and influential.

A PL in which variable names are references to objects in memory is said to have ***reference semantics***. All FPLs and most OOPLs have reference semantics.

Note that reference semantics is not the same as “pointers” in languages such as Pascal and C. A pointer variable stands for a location in memory and therefore has value semantics; it just so happens that the location is used to store the address of another object.

**Lambda.** LISP uses “lambda expressions”, based on Church’s $\lambda$-calculus, to denote functions. For example, the function that squares its argument is written

```
(lambda (x) (* x x))
```

by analogy to Church’s $f = \lambda x \,.\, x^2$. We can apply a lambda expression to an argument to obtain the value of a function application. For example, the expression

```
((lambda (x) (* x x)) 4)
```

yields the value 16.

However, the lambda expression itself cannot be evaluated. Consequently, LISP had to resort to programming tricks to make higher order functions work. For example, if we want to pass the squaring function as an argument to another function, we must wrap it up in a “special form” called `function`:

```
(f (function (lambda (x) (* x x))) ....)
```

Similar complexities arise when a function returns another function as a result.

**Listing 7:** Static and Dynamic Binding

```
int x = 4;          // 1
void f()
{
  printf("%d", x);
}
void main ()
{
  int x = 7;          // 2
  f();
}
```

**Dynamic Scoping.** Dynamic scoping was an "accidental" feature of LISP: it arose as a side-effect of the implementation of the look-up table for variable values used by the interpreter. The C-like program in Listing 7 illustrates the difference between static and dynamic scoping. In C, the variable $x$ in the body of the function $f$ is a use of the global variable $x$ defined in the first line of the program. Since the value of this variable is 4, the program prints 4. (Do not confuse dynamic scoping with dynamic binding!)

A LISP interpreter constructs its environment as it interprets. The environment behaves like a stack (last in, first out). The initial environment is empty, which we denote by $\langle\rangle$. After interpreting the LISP equivalent of the line commented with "1", the environment contains the global binding for $x$: $\langle x = 4\rangle$. When the interpreter evaluates the function `main`, it inserts the local $x$ into the environment, obtaining $\langle x = 7, x = 4\rangle$. The interpreter then evaluates the call $f()$; when it encounters $x$ in the body of $f$, it uses the first value of $x$ in the environment and prints 7.

Although dynamic scoping is natural for an interpreter, it is inefficient for a compiler. Interpreters are slow anyway, and the overhead of searching a linear list for a variable value just makes them slightly slower still. A compiler, however, has more efficient ways of accessing variables, and forcing it to maintain a linear list would be unacceptably inefficient. Consequently, early LISP systems had an unfortunate discrepancy: the interpreters used dynamic scoping and the compilers used static scoping. Some programs gave one answer when interpreted and another answer when compiled!

**Exercise 26.** Describe a situation in which dynamic scoping is useful. □

**Interpretation.** LISP was the first major language to be interpreted. Originally, the LISP interpreter behaved as a calculator: it evaluated expressions entered by the user, but its internal state did not change. It was not long before a form for defining functions was introduced (originally called `define`, later changed to `defun`) to enable users to add their own functions to the list of built-in functions.

A LISP program has no real structure. On paper, a program is a list of function definitions; the functions may invoke one another with either direct or indirect recursion. At run-time, a program is the same list of functions, translated into internal form, added to the interpreter.

The current dialect of LISP is called Common LISP (Steele et al. 1990). It is a much larger and more complex language than the original LISP and includes many features of Scheme (described below). Common LISP provides static scoping with dynamic scoping as an option.

**Listing 8:** Defining `car`

```
        (cond
          ((eq (car expr) 'car)
            (car (cadr expr)) )
        ....
```

**Listing 9:** Factorial with functions

```
        (define factorial
          (lambda (n) (if (= n 0)
            1
            (* n (factorial (- n 1))))))
```

**Exercise 27.** The LISP interpreter, written in LISP, contains expressions such as the one shown in Listing 8. We might paraphrase this as: "if the `car` of the expression that we are currently evaluating is `car`, the value of the expression is obtained by taking the `car` of the `cadr` (that is, the second term) of the expression". How much can you learn about a language by reading an interpreter written in the language? What can you ***not*** learn? □

## 5.2 Scheme

Scheme was designed by Guy L. Steele Jr. and Gerald Jay Sussman (1975). It is very similar to LISP in both syntax and semantics, but it corrects some of the errors of LISP and is both simpler and more consistent.

The starting point of Scheme was an attempt by Steele and Sussman to understand Carl Hewitt's theory of ***actors*** as a model of computation. The model was object oriented and influenced by Smalltalk (see Section 6.2). Steele and Sussman implemented the actor model using a small LISP interpreter. The interpreter provided lexical scoping, a `lambda` operation for creating functions, and an `alpha` operation for creating actors. For example, the factorial function could be represented either as a function, as in Listing 9, or as an actor, as in Listing 10. Implementing the interpreter brought an odd fact to light: the interpreter's code for handling `alpha` was identical to the code for handling `lambda`! This indicated that closures — the objects created by evaluating `lambda` — were useful for both high order functional programming and object oriented programming (Steele 1996).

LISP ducks the question "what is a function?" It provides lambda notation for functions, but a lambda expression can only be applied to arguments, not evaluated itself. Scheme provides an answer to this question: the value of a function is a ***closure***. Thus in Scheme we can write both

```
        (define num 6)
```

which binds the value 6 to the name `num` and

**Listing 10:** Factorial with actors

```
        (define actorial
          (alpha (n c) (if (= n 0)
            (c 1)
            (actorial (- n 1) (alpha (f) (c (* f n)))))))
```

**Listing 11:** Differentiating in Scheme

```
(define derive (lambda (f dx)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x)) dx))))
(define square (lambda (x) (* x x)))
(define Dsq (derive sq 0.001))
```

```
(define square (lambda (x) (* x x)))
```

which binds the squaring function to the name `square`. (Scheme actually provides an abbreviated form of this definition, to spare programmers the trouble of writing `lambda` all the time, but the form shown is accepted by the Scheme compiler and we use it in these notes.)

Since the Scheme interpreter accepts a series of definitions, as in LISP, it is important to understand the effect of the following sequence:

```
(define n 4)
(define f (lambda () n))
(define n 7)
(f)
```

The final expression calls the function `f` that has just been defined. The value returned is the value of `n`, but which value, 4 or 7? If this sequence could be written in LISP, the result would be 7, which is the value of `n` in the environment when `(f)` is evaluated. Scheme, however, uses ***static scoping***. The closure created by evaluating the definition of `f` includes all name bindings in effect at ***the time of definition***. Consequently, a Scheme interpreter yields 4 as the value of `(f)`. The answer to the question “what is a function?” is “a function is an expression (the body of the function) and an environment containing the values of all variables accessible at the point of definition”.

Closures in Scheme are ordinary values. They can be passed as arguments and returned by functions. (Both are possible in LISP, but awkward because they require special forms.)

**Example 14: Differentiating.** Differentiation is a function that maps functions to functions. Approximately (Abelson and Sussman 1985):

$$D\,f(x) \quad = \quad \frac{f(x+dx)-f(x)}{dx}$$

We define the Scheme functions shown in Listing 11. After these definitions have been evaluated, `Dsq` is, effectively, the function

$$\begin{aligned} f(x) &= \frac{(x+0.001)^2 - x^2}{0.001} \\ &= 2x + \cdots \end{aligned}$$

We can apply `Dsq` like this: (`->`” is the Scheme prompt):

```
-> (Dsq 3)
6.001
```

□

Scheme avoids the problem of incompatibility between interpretation and compilation by being statically scoped, whether it is interpreted or compiled. The interpreter uses a more elaborate data

**Listing 12:** Banking in Scheme

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
      (sequence (set!  balance (- balance amount))
        balance)
      ("Insufficient funds"))
  (define (deposit amount)
    (set!  balance (+ balance amount))
      balance)
  (define (dispatch m)
    (cond
      ((eq?  m 'withdraw) withdraw)
      ((eq?  m 'deposit) deposit)
      (else (error "Unrecognized transaction" m))))
  dispatch)
```

**Listing 13:** Using the account

```
-> ((acc 'withdraw) 50)
50
-> ((acc 'withdraw) 100)
Insufficient funds
-> ((acc 'deposit) 100)
150
```

structure for storing values of local variables to obtain the effect of static scoping. In addition to full support for high order functions, Scheme introduced ***continuations***.

Although Scheme is primarily a functional language, side-effects are allowed. In particular, `set!` changes the value of a variable. (The "!" is a reminder that `set!` has side-effects.)

**Example 15: Banking in Scheme.** The function shown in Listing 12 shows how side-effects can be used to define an object with changing state (Abelson and Sussman 1985, page 173). The following dialog shows how banking works in Scheme. The first step is to create an account.

```
-> (define acc (make-account 100))
```

The value of `acc` is a closure consisting of the function `dispatch` together with an environment in which `balance = 100`. The function `dispatch` takes a single argument which must be `withdraw` or `deposit`; it returns one of the functions `withdraw` or `deposit` which, in turn, takes an amount as argument. The value returned is the new balance. Listing 13 shows some simple applications of acc. The quote sign (`'`) is required to prevent the evaluation of `withdraw` or `deposit`. □

This example demonstrates that with higher order functions and control of state (by side-effects) we can obtain a form of OOP. The limitation of this approach, when compared to OOPLs such as Simula and Smalltalk (described in Section 6) is that we can define only one function at a time. This function must be used to dispatch messages to other, local, functions.

## 5.3 SASL

SASL (St. Andrew's Symbolic Language) was introduced by David Turner (1976). It has an Algol-like syntax and is of interest because the compiler translates the source code into a combinator expression which is then processed by graph reduction (Turner 1979). Turner subsequently designed KRC (Kent Recursive Calculator) (1981) and Miranda (1985), all of which are implemented with combinator reduction.

Combinator reduction implements call by name (the default method for passing parameters in Algol 60) but with an optimization. If the parameter is not needed in the function, it is not evaluated, as in Algol 60. If it is needed one or more times, it is evaluated exactly once. Since SASL expressions do not have side-effects, evaluating an expression more than once will always give the same result. Thus combinator reduction is (in this sense) the most efficient way to pass parameters to functions. Evaluating an expression only when it is needed, and never more than once, is called ***call by need*** or ***lazy evaluation***.

The following examples use SASL notation. The expression `x::xs` denotes a list with first element (head) `x` and remaining elements (tail) `xs`. The definition

```
nums(n) = n::nums(n+1)
```

apparently defines an infinite list:

```
nums(0) = 0::nums(1) = 0::1::nums(2) = ....
```

The function `second` returns the second element of a list. In SASL, we can define it like this:

```
second (x::y::xs) = y
```

Although `nums(0)` is an "infinite" list, we can find its second element in SASL:

```
second(nums(0)) = second(0::nums(1)) = second(0::1::nums(2)) = 1
```

This works because SASL evaluates a parameter only when its value is needed for the calculation to proceed. In this example, as soon as the argument of `second` is in the form `0::1::....`, the required result is known.

Call by need is the only method of passing arguments in SASL but it occurs as a special case in other languages. If we consider `if` as a function, so that the expression

```
if P then X else Y
```

is a fancy way of writing `if(P,X,Y)`, then we see that `if` must use call by need for its second and third arguments. If it did not, we would not be able to write expressions such as

```
if x = 0 then 1 else 1/x
```

In C, the functions `&&` (AND) and || (OR) are defined as follows:

$$\begin{aligned} X \;\texttt{\&\&}\; Y &\equiv \texttt{if}\; X \;\texttt{then}\; Y \;\texttt{else false} \\ X \mid\mid Y &\equiv \texttt{if}\; X \;\texttt{then true else}\; Y \end{aligned}$$

These definitions provide the effect of lazy evaluation and allow us to write expressions such as

```
if (p != NULL && p->f > 0) ....
```

## 5.4 SML

SML (Milner, Tofte, and Harper 1990; Milner and Tofte 1991) was designed as a "metalanguage" (ML) for reasoning about programs as part of the Edinburgh Logic for Computable Functions

**Listing 14:** Function composition

```
- infix o;
- fun (f o g) x = g (f x);
val o = fn :  (’a -> ’b) * (’b -> ’c) -> ’a -> ’c
- val quad = sq o sq;
val quad = fn :  real -> real
- quad 3.0;
val it = 81.0 :  real
```

(LCF) project. The language survived after the rest of the project was abandoned and became “standard” ML, or SML. The distinguishing feature of SML is that it is statically typed in the sense of Section 3.3 and that most types can be inferred by the compiler.

In the following example, the programmer defines the factorial function and SML responds with its type. The programmer then tests the factorial function with argument 6; SML assigns the result to the variable `it`, which can be used in the next interaction if desired. SML is run interactively, and prompts with “`-`”.

```
- fun fac n = if n = 0 then 1 else n * fac(n-1);
val fac = fn :  int -> int
- fac 6;
val it = 720 :  int
```

SML also allows function declaration *by cases*, as in the following alternative declaration of the factorial function:

```
- fun fac 0 = 1
= | fac n = n * fac(n-1);
val fac = fn :  int -> int
- fac 6;
val it = 720 :  int
```

Since SML recognizes that the first line of this declaration is incomplete, it changes the prompt to “`=`” on the second line. The vertical bar “`|`” indicates that we are declaring another “case” of the declaration.

Each case of a declaration by cases includes a *pattern*. In the declaration of `fac`, there are two patterns. The first, `0`, is a *constant pattern*, and matches only itself. The second, `\tt` n, is a *variable pattern*, and matches any value of the appropriate type. Note that the definition `fun sq x = x * x;` would fail because SML cannot decide whether the type of `x` is `int` or `real`.

```
- fun sq x:real = x * x;
val sq = fn :  real -> real
- sq 17.0;
val it = 289.0 :  real
```

We can pass functions as arguments to other functions. The function `o` (intended to resemble the small circle that mathematicians use to denote functional composition) is built-in, but even if it wasn’t, we could easily declare it and use it to build the fourth power function, as in Listing 14. The symbols `’a`, `’b`, and `’c` are type names; they indicate that SML has recognized `o` as a polymorphic function.

The function `hasfactor` defined in Listing 15 returns `true` if its first argument is a factor of

**Listing 15:** Finding factors

```
        - fun hasfactor f n = n mod f = 0;
        val hasfactor fn :  int -> int -> bool
        - hasfactor 3 9;
        val it = true :  bool
```

**Listing 16:** A function with one argument

```
        - val even = hasfactor 2;
        val even = fn :  int -> bool;
        - even 6;
        val it = true :  bool
```

its second argument. All functions in SML have *exactly one* argument. It might appear that `hasfactor` has two arguments, but this is not the case. The declaration of `hasfactor` introduces two functions, as shown in Listing 16. Functions like `hasfactor` take their arguments one at a time. Applying the first argument, as in `hasfactor 2`, yields a new function. The trick of applying one argument at a time is called “currying”, after the American logician Haskell Curry. It may be helpful to consider the types involved:

```
        hasfactor :  int -> int -> bool
        hasfactor 2 :  int -> bool
        hasfactor 2 6 :  bool
```

The following brief discussion, adapted from (Åke Wikström 1987), shows how functions can be used to build a programmer’s toolkit. The functions here are for list manipulation, which is a widely used example but not the only way in which a FPL can be used. We start with a list generator, defined as an infix operator.

```
        - infix --;
        - fun (m -- n) = if m < n then m ::  (m+1 -- n) else [];
        val -- = fun :  int * int -> int list
        - 1 -- 5;
        val it = [1,2,3,4,5] :  int list
```

The functions `sum` and `prod` in Listing 17 compute the sum and product, respectively, of a list of integers. We note that `sum` and `prod` have a similar form. This suggests that we can abstract the

**Listing 17:** Sums and products

```
        - fun sum [] = 0
        = | sum (x::xs) = x + sum xs;
        val sum = fn :  int list -> int
        - fun prod [] = 1
        = | prod (x::xs) = x * prod xs;
        val prod = fn :  int list -> int
        - sum (1 -- 5);
        val it = 15 :  int
        - prod (1 -- 5);
        val it = 120 :  int
```

**Listing 18:** Using `reduce`

```
- fun sum xs = reduce add 0 xs;
val sum = fn :  int list -> list
- fun prod xs = reduce mul 1 xs;
val prod = fn :  int list -> list
```

common features into a function `reduce` that takes a binary function, a value for the empty list, and a list. We can use `reduce` to obtain one-line definitions of `sum` and `prod`, as in Listing 18. The idea of processing a list by recursion has been captured in the definition of `reduce`.

```
- fun reduce f u [] = u
= | reduce f u (x::xs) = f x (reduce f u xs);
val reduce = fn :  (’a -> ’b -> ’b) -> ’b -> ’a list -> ’b
```

We can also define a sorting function as

```
- val sort = reduce insert nil;
val sort = fn :  int list -> int list
```

where `insert` is the function that inserts a number into an ordered list, preserving the ordering:

```
- fun insert x:int [] = x::[]
| insert x (y::ys) = if x <= y then x::y::ys else y::insert x ys;
val insert = fn :  int -> int list -> int list
```

## 5.5 Other Functional Languages

We have not discussed FP, the notation introduced by Backus (1978). Although Backus’s speech (and subsequent publication) turned many people in the direction of functional programming, his ideas for the design of FPLs were not widely adopted.

Since 1978, FPLs have split into two streams: those that provide mainly “eager evaluation” (that is, call by value), following LISP and Scheme, and those that provide mainly “lazy evaluation”, following SASL. The most important language in the first category is still SML, while Haskell appears to be on the way to becoming the dominant FPL for teaching and other applications.

# 6 The Object Oriented Paradigm

> A fundamental feature of our understanding of the world is that we organize our experience as a number of distinct ***object*** (tables and chairs, bank loans and algebraic expressions, . . .) . . . . When we wish to solve a problem on a computer, we often need to construct within the computer a ***model*** of that aspect of the real or conceptual world to which the solution of the problem will be applied. (Hoare 1968)

Object Oriented Programming (OOP) is the currently dominant programming paradigm. For many people today, "programming" means "object oriented programming". Nevertheless, there is a substantial portion of the software industry that has ***not*** adopted OOP, and there remains widespread misunderstanding about what OOP actually is and what, if any, benefits it has to offer.

## 6.1 Simula

Many programs are computer simulations of the real world or a conceptual world. Writing such programs is easier if there is a correspondence between objects in the world and components of the program.

Simula originated in the Norwegian Computing Centre in 1962. Kristen Nygaard proposed a language for simulation to be developed by himself and Ole-Johan Dahl (1978). Key insights were developed in 1965 following experience with Simula I:

▷ the purpose of the language was to model ***systems***;

▷ a system is a collection of interacting ***processes***;

▷ a process can be represented during program execution by multiple procedures each with its own Algol-style ***stacks***.

The main lessons of Simula I were:

▷ the distinction between a program text and its execution;

▷ the fact that data and operations belong together and that most useful programming constructs contain both.

Simula 67 was a general purpose programming language that incorporated the ideas of Simula I but put them into a more general context. The basic concept of Simula 67 was to be "***classes*** of ***objects***". The major innovation was "block prefixing" (Nygaard and Dahl 1978).

Prefixing emerged from the study of queues, which are central to discrete-event simulations. The queue mechanism (a list linked by pointers) can be split off from the elements of the queue (objects such as trucks, buses, people, and so on, depending on the system being simulated). Once recognized, the prefix concept could be used in any context where common features of a collection of classes could be abstracted in a prefixed block or, as we would say today, a ***superclass***.

Dahl (1978, page 489) has provided his own analysis of the role of blocks in Simula.

▷ Deleting the procedure definitions and final statement from an Algol block gives a pure data record.

▷ Deleting the final statement from an Algol block, leaving procedure definitions and data declarations, gives an ***abstract data object***.

**Listing 19:** A Simula Class

```
        class Account (real balance);
        begin
          procedure Deposit (real amount)
            balance := balance + amount;
          procedure Withdraw (real amount)
            balance := balance - amount;
        end;
```

- ▷ Adding coroutine constructs to Algol blocks provides quasi-parallel programming capabilities.
- ▷ Adding a prefix mechanism to Algol blocks provides an abstraction mechanism (the class hierarchy).

All of these concepts are subsumed in the Simula ***class***.

Listing 19 shows a simple Simula-style class, with modernized syntax. Before we use this class, we must declare a reference to it and then instantiate it on the heap:

```
ref (Account) myAccount;
MyAccount := new Account(1000);
```

Note that the class is syntactically more like a procedure than a data object: it has an (optional) formal parameter and it must be called with an actual parameter. It differs from an ordinary Algol procedure in that its AR remains on the heap after the invocation.

To inherit from `Account` we write something like:

```
Account class ChequeAccount (real amount);
  . . . .
```

This explains the term "prefixing" — the declaration of the child class is prefixed by the name of its parent. Inheritance and subclasses introduced a new programming methodology, although this did not become evident for several years.

Simula provides:

- ▷ ***coroutines*** that permit the simulation of concurrent processes;
- ▷ ***multiple stacks***, needed to support coroutines;
- ▷ ***classes*** that combine data and a collection of functions that operate on the data;
- ▷ ***prefixing*** (now known as ***inheritance***) that allows a specialized class to be derived from a general class without unnecessary code duplication;
- ▷ a ***garbage collector*** that frees the programmer from the responsibility of deallocating storage.

Simula itself had an uneven history. It was used more in Europe than in North America, but it never achieved the recognition that it deserved. This was partly because there were few Simula compilers and the good compilers were expensive.

On the other hand, the legacy that Simula left is considerable: a new paradigm of programming.

**Exercise 28.** Coroutines are typically implemented by adding two new statements to the language: `suspend` and `resume`. A `suspend` statement, executed within a function, switches control to a scheduler which stores the value of the program counter somewhere and continues the execution of another function. A `resume` statement has the effect of restarting a previously `suspend`ed function.

At any one time, exactly one function is executing. What features might a concurrent language provide that are not provided by coroutines? What additional complications are introduced by concurrency? □

## 6.2 Smalltalk

Smalltalk originated with Alan Kay's reflections on the future of computers and programming in the late 60s. Kay (1996) was influenced by LISP, especially by its one-page metacircular interpreter, and by Simula. It is interesting to note that Kay gave a talk about his ideas at MIT in November 1971; the talk inspired Carl Hewitt's work on his Actor model, an early attempt at formalizing objects. In turn, Sussman and Steele wrote an interpreter for Actors that eventually became Scheme (see Section 5.2). The cross-fertilization between OOPL and FPL that occurred during these early days has, sadly, not continued.

The first version of Smalltalk was implemented by Dan Ingalls in 1972, using BASIC (!) as the implementation language (Kay 1996, page 533). Smalltalk was inspired by Simula and LISP; it was based on six principles (Kay 1996, page 534).

1. Everything is an ***object***.
2. Objects communicate by sending and receiving ***messages*** (in terms of objects).
3. Objects have their own memory (in terms of objects).
4. Every object is an ***instance*** of a ***class*** (which must be an object).
5. The class holds the shared ***behaviour*** for its instances (in the form of objects in a program list).
6. To [evaluate] a program list, control is passed to the first object and the remainder is treated as its message.

Principles 1–3 provide an "external" view and remained stable as Smalltalk evolved. Principles 4–6 provide an "internal" view and were revised following implementation experience. Principle 6 reveals Kay's use of LISP as a model for Smalltalk — McCarthy had described LISP with a one-page meta-circular interpreter; one of Kay's goals was to do the same for Smalltalk.

Smalltalk was also strongly influenced by Simula. However, it differs from Simula in several ways:

- ▷ Simula distinguishes primitive types, such as `integer` and `real`, from class types. In Smalltalk, "everything is an object".
- ▷ In particular, classes are objects in Smalltalk. To create a new instance of a class, you send a message to it. Since a class object must belong to a class, Smalltalk requires metaclasses.
- ▷ Smalltalk effectively eliminates passive data. Since objects are "active" in the sense that they have methods, and everything is an object, there are no data primitives.
- ▷ Smalltalk is a complete environment, not just a compiler. You can edit, compile, execute, and debug Smalltalk programs without ever leaving the Smalltalk environment.

The "block" is an interesting innovation of Smalltalk. A ***block*** is a sequence of statements that can be passed as a parameter to various control structures and behaves rather like an object. In the Smalltalk statement

```
        10 timesRepeat: [Transcript nextPutAll: ' Hi!']
```

the receiver is `10`, the message is `timesRepeat:`, and `[Transcript nextPutAll: ' Hi!']` is the parameter of the message.

Blocks may have parameters. One technique for iteration in Smalltalk is to use a `do:` message which executes a block (its parameter) for each component of a collection. The component is passed to the block as a parameter. The following loop determines whether `"Mika"` occurs in the current object (assumed to be a collection):

```
        self do: [:item | item = "Mika" ifTrue: [^true]].
        ^false
```

The first occurrence of `item` introduces it as the name of the formal parameter for the block. The second occurrence is its use. Blocks are closely related to closures (Section 5.2). The block `[:x | E]` corresponds to $\lambda x \,.\, E$.

The first practical version of Smalltalk was developed in 1976 at Xerox Palo Alto Research Center (PARC). The important features of Smalltalk are:

- ▷ everything is an object;
- ▷ an object has private data and public functions;
- ▷ objects collaborate by exchanging "messages";
- ▷ every object is a member of a class;
- ▷ there is an inheritance hierarchy (actually a tree) with the class `Object` as its root;
- ▷ all classes inherit directly or indirectly from the class `Object`;
- ▷ blocks;
- ▷ coroutines;
- ▷ garbage collection.

**Exercise 29.** Discuss the following statements (Gelernter and Jagannathan 1990, page 245).

**1.** [Data objects are replaced by program structures.] Under this view of things, particularly under Smalltalk's approach, there are no passive objects to be acted ***upon***. Every entity is an ***active*** agent, a little program in itself. This can lead to counter-intuitive interpretations of simple operations, and can be destructive to a natural sense of hierarchy.

**2.** Algol 60 gives us a representation for procedures, but no direct representation for a procedure ***activation***. Simula and Smalltalk inherit this same limitation: they give us representations for ***classes***, but no direct representations for the objects that are instantiated from classes. Our ability to create single objects and to specify (constant) object "values" is compromised. □

**Exercise 30.** In Smalltalk, an object is an instance of a class. The class is itself an object, and therefore is a member of a class. A class whose instances are classes is called a ***metaclass***. Do these definitions imply an infinite regression? If so, how could the infinite regression by avoided? Is there a class that is a member of itself? □

## 6.3 CLU

CLU is not usually considered an OOPL because it does not provide any mechanisms for incremental modification. However:

> The resulting programming methodology is object-oriented: programs are developed by thinking about the objects they manipulate and then inventing a modular structure based on these objects. Each type of object is implemented by its own program module. (Liskov 1996, page 475)

Parnas (1972) introduced the idea of "information hiding": a program module should have a public interface and a private implementation. Liskov and Zilles (1974) introduced CLU specifically to support this style of programming.

An ***abstract data type*** (ADT) specifies a type, a set of operations that may be performed on instances of the type, and the effects of these operations. The implementation of an ADT provides the actual operations that have the specified effects and also prevents programmers from doing anything else. The word "abstract" in CLU means "user defined", or not built in to the language:

> I referred to the types as "abstract" because they are not provided directly by a programming languages but instead must be implemented by the user. An abstract type is abstract in the same way that a procedure is an abstract operation. (Liskov 1996, page 473)

This is not a universally accepted definition. Many people use the term ADT in the following sense:

> An ***abstract data type*** is a system consisting of three constituents:
>
> 1. some sets of objects;
> 2. a set of syntactic descriptions of the primitive functions;
> 3. a semantic description — that is, a sufficiently complete set of relationships that specify how the functions interact with each other.
>
> (Martin 1986)

The difference is that an ADT in CLU provides a particular implementation of the type (and, in fact, the implementation must be unique) whereas Martin's definition requires only a specification. A stack ADT in CLU would have functions called `push` and `pop` to modify the stack, but a stack ADT respecting Martin's definition would define the key property of a stack (the sequence `push(); pop()` has no effect) but would ***not*** provide an implementation of these functions.

Although Simula influenced the design of CLU, it was seen as deficient in various ways. Simula:

- ▷ does not provide encapsulation: clients could directly access the variables, as well as the functions, of a class;
- ▷ does not provide "type generators" or, as we would say now, "generic types" ("templates" in C++);
- ▷ associates operations with objects rather than types;
- ▷ handles built-in and user-defined types differently — for example, instances of user-defined classes are always heap-allocated.

CLU's computational model is similar to that of LISP: names are references to heap-allocated objects. The stack-only model of Algol 60 was seen as too restrictive for a language supporting data abstractions. The reasons given for using a heap include:

**Listing 20:** A CLU Cluster

```
intset = cluster is create, insert, delete, member, size, choose
  rep = array[int]
  create = proc () returns (cvt)
     return (rep$new())
    end create
  insert = proc (s:  intset, x:  int)
    if ~member(s, x) then rep$addh(down(s), x) end
    end insert
  member = proc (s:  cvt, x:  int) returns (bool)
    return (getind(s, x) ≤ rep$high(s))
  end member
....
end intset
```

- Declarations are easy to implement because, for all types, space is allocated for a pointer. Stack allocation breaks encapsulation because the size of the object (which should be an implementation "secret") must be known at the point of declaration.
- Variable declaration is separated from object creation. This simplifies safe initialization.
- Variable and object lifetimes are not tied. When the variable goes out of scope, the object continues to exist (there may be other references to it).
- The meaning of assignment is independent of type. The statement $x := E$ means simply that $x$ becomes a reference (implemented as a pointer, of course) to $E$. There is no need to worry about the semantics of copying an object. Copying, if needed, should be provided by a member function of the class.
- Reference semantics requires garbage collection. Although garbage collection has a run-time overhead, efficiency was not a primary goal of the CLU project. Furthermore, garbage collection eliminates memory leaks and dangling pointers, notorious sources of obscure errors.

CLU is designed for programming with ADTs. Associated with the interface is an implementation that defines a representation for the type and provides bodies for the functions.

Listing 20 shows an extract from a cluster that implements a set of integers (Liskov and Guttag 1986, page 63). The components of a cluster, such as `intset`, are:

1. A ***header*** that gives the name of the type being implemented (`intset`) and the names of the operations provided (`create`, `insert`, . . . .).
2. A definition of the representation type, introduced by the keyword `rep`. An `intset` is represented by an array of integers.
3. Implementations of the functions mentioned in the header and, if necessary, auxiliary functions that are not accessible to clients.

The abstract type `intset` and the representation type `array[int]`, referred to as `rep`, are distinct. CLU provides special functions `up`, which converts the representation type to the abstract type, and `down`, which converts the abstract type to the representation type. For example, the function `insert` has a parameter `s` of type `intset` which is converted to the representation type so that it can be treated as an array.

It often happens that the client provides an abstract argument that must immediately be converted to the representation type; conversely, many functions create a value of the representation type but must return a value of the abstract type. For both of these cases, CLU provides the keyword `cvt` as an abbreviation. In the function `create` above, `cvt` converts the new instance of `rep` to `intset` before returning, and in the function `member`, the formal parameter `s:cvt` indicates that the client will provide an `intset` which will be treated in the function body as a `rep`.

All operations on abstract data in CLU contain the type of the operator explicitly, using the syntax `type$operation`.

CLU draws a distinction between mutable and immutable objects. An object is ***mutable*** if its value can change during execution, otherwise it is ***immutable***. For example, integers are immutable but arrays are mutable (Liskov and Guttag 1986, pages 19–20).

The mutability of an object depends on its type and, in particular, on the operations provided by the type. Clearly, an object is mutable if its type has operations that change the object. After executing

```
x:  T = intset$create()
y:  T
y := x
```

the variables `x` and `y` refer to the same object. If the object is immutable, the fact that it is shared is undetectable by the program. If it is mutable, changes made via the reference `x` will be visible via the reference `y`. Whether this is desirable depends on your point of view.

The important contributions of CLU include:

- modules (“clusters”);
- safe encapsulation;
- distinction between the abstract type and the representation type;
- names are references, not values;
- the distinction between mutable and immutable objects;
- analysis of copying and comparison of ADTs;
- generic clusters (a cluster may have parameters, notably type parameters);
- exception handling.

Whereas Simula provides tools for the programmer and supports a methodology, CLU provides the same tools and enforces the methodology. The cost is an increase in complexity. Several keywords (`rep`, `cvt`, `down`, and `up`) are needed simply to maintain the distinction between abstract and representation types. In Simula, three concepts — procedures, classes, and records — are effectively made equivalent, resulting in significant simplification. In CLU, procedures, records, and clusters are three different things.

The argument in favour of CLU is that the compiler will detect encapsulation and other errors. But is it the task of a compiler to prevent the programmer doing things that might be completely safe? Or should this role be delegated to a style checker, just as C programmers use both `cc` (the C compiler) and `lint` (the C style checker)?

The designers of CLU advocate ***defensive programming***:

> Of course, checking whether inputs [i.e., parameters of functions] are in the permitted subset of the domain takes time, and it is tempting not to bother with the checks, or to use them only while debugging, and suppress them during production. This is generally an unwise practice. It is better to develop the habit of *defensive programming*, that it writing each procedure to defend itself against errors. . . .
>
> Defensive programming makes it easier to debug programs. . . . (Liskov and Guttag 1986, page 100)

**Exercise 31.** Do you agree that defensive programming is a better habit than the alternative suggested by Liskov and Guttag? Can you propose an alternative strategy? □

## 6.4 C++

C++ was developed at Bell Labs by Bjarne Stroustrup (1994). The task assigned to Stroustrup was to develop a new systems PL that would replace C. C++ is the result of two major design decisions: first, Stroustrup decided that the new language would be adopted only if it was compatible with C; second, Stroustrup's experience of completing a Ph.D. at Cambridge University using Simula convinced him that object orientation was the correct approach but that efficiency was essential for acceptance. C++ is widely used but has been sharply criticized (Sakkinen 1988; Sakkinen 1992; Joyner 1992).

C++:

- is almost a superset of C (that is, there are only a few C constructs that are not accepted by a C++ compiler);
- is a hybrid language (i.e., a language that supports both imperative and OO programming), not a "pure" OO language;
- emphasizes the stack rather than the heap, although both stack and heap allocation is provided;
- provides multiple inheritance, genericity (in the form of "templates"), and exception handling;
- does not provide garbage collection.

**Exercise 32.** Explain why C++ is the most widely-used OOPL despite its technical flaws. □

## 6.5 Eiffel

Software Engineering was a key objective in the design of Eiffel.

> Eiffel embodies a "certain idea" of software construction: the belief that it is possible to treat this task as a serious engineering enterprise, whose goal is to yield quality software through the careful production and continuous development of parameterizable, scientifically specified, reusable components, communicating on the basis of clearly defined contracts and organized in systematic multi-criteria classifications. (Meyer 1992)

Eiffel borrows from Simula 67, Algol W, Alphard, CLU, Ada, and the specification language Z. Like some other OOPLs, it provides multiple inheritance. It is notable for its strong typing, an unusual exception handling mechanism, and assertions. By default, object names are references; however, Eiffel also provides "expanded types" whose instances are named directly.

**Listing 21:** Finding a square root

```
double sqrt (double x)
{
  if (x ≥ 0.0)
    return .... // √x
  else
    ??
}
```

**Listing 22:** A contract for square roots

```
sqrt (x: real): real is
  require x ≥ 0.0
  Result := .... -- √x
  ensure abs(Result * Result / x - 1.0) ≤ 1e-6
end
```

Eiffel does not have: global variables; enumerations; subranges; `goto`, `break`, or `continue` statements; procedure variables; casts; or pointer arithmetic. Input/output is defined by libraries rather than being built into the language.

The main features of Eiffel are as follows. Eiffel:

- ▷ is strictly OO: all functions are defined as methods in classes;
- ▷ provides multiple inheritance and repeated inheritance;
- ▷ supports "programming by contract" with assertions;
- ▷ has an unusual exception handling mechanism;
- ▷ provides generic classes;
- ▷ may provide garbage collection.

### 6.5.1 Programming by Contract

The problem with writing a function to compute square roots is that it is not clear what to do if the argument provided by the caller is negative. The solution adopted in Listing 21 is to allow for the possibility that the argument might be negative.

- ▷ This approach, applied throughout a software system, is called ***defensive programming*** (Section 6.3).
- ▷ Testing arguments adds overhead to the program.
- ▷ Most of the time spent testing is wasted, because programmers will rarely call `sqrt` with a negative argument.
- ▷ The failure action, indicated by "`??`" in Listing 21, is hard to define. In particular, it is undesirable to return a funny value such as $-999.999$, or to send a message to the console (there may not be one), or to add another parameter to indicate success/failure. The only reasonable possibility is some kind of exception.

Meyer's solution is to write `sqrt` as shown in Listing 22. The `require`/`ensure` clauses constitute a ***contract*** between the function and the caller. In words:

**Listing 23:** Eiffel contracts

```
class Parent
  feature
    f() is
      require $R_p$
      ensure $E_p$
  end
....
class Child inherit Parent
  feature
    f() is
      require $R_c$
      ensure $E_c$
....
```

> If the actual parameter supplied, `x`, is positive, the result returned will be approximately $\sqrt{\mathtt{x}}$.

If the argument supplied is negative, the contract does not constrain the function in any way at all: it can return -999.999 or any other number, terminate execution, or whatever. The caller therefore has a responsibility to ensure that the argument is valid.

The advantages of this approach include:

- Many tests become unnecessary.
- Programs are not cluttered with tests for conditions that are unlikely to arise.
- The `require` conditions can be checked dynamically during testing and (perhaps) disabled during production runs. (Hoare considers this policy to be analogous to wearing your life jacket during on-shore training but leaving it behind when you go to sea.)
- `Require` and `ensure` clauses are a useful form of documentation; they appear in Eiffel interfaces.

Functions in a child class must provide a contract at least as strong as that of the corresponding function in the parent class, as shown in Listing 23. We must have $R_p \Rightarrow R_c$ and $E_c \Rightarrow E_p$. Note the directions of the implications: the `ensure` clauses are "covariant" but the `require` clauses are "contravariant". The following analogy explains this reversal.

A pet supplier offers the following contract: "if you send at least 200 mk, I will send you an animal". The pet supplier has a subcontractor who offers the following contract: "if you send me at least 150 mk, I will send you a dog". These contracts respect the Eiffel requirements: the contract offered by the subcontractor has a weaker requirement (150 mk is less than 200 mk) but promises more (a dog is more specific than an animal). Thus the contractor can use the subcontractor when a dog is required and make a profit of 50 mk.

Eiffel achieves the subcontract specifications by requiring the programmer to define

$$
\begin{aligned}
R_c &\equiv R_p \vee \cdots \\
E_c &\equiv E_p \wedge \cdots
\end{aligned}
$$

**Listing 24:** Repeated inheritance in Eiffel

```
class House
  feature
    address:  String
    value:  Money
end
class Residence inherit House
  rename value as residenceValue
end
class Business inherit House
  rename value as businessValue
end
class HomeBusiness inherit Residence Business
....
end
```

The $\vee$ ensures that the child's requirement is weaker than the parent's requirement, and the $\wedge$ ensures that the child's commitment is stronger than the parent's commitment.

In addition to `require` and `ensure` clauses, Eiffel has provision for writing class ***invariants***. A class invariant is a predicate over the variables of an instance of the class. It must be true when the object is created and after each function of the class has executed.

Eiffel programmers are not obliged to write assertions, but it is considered good Eiffel "style" to include them. In practice, it is sometimes difficult to write assertions that make non-trivial claims about the objects. Typical contracts say things like: "after you have added an element to this collection, it contains one more element". Since assertions may contain function calls, it is possible in principle to say things like: "this structure is a heap (in the sense of heap sort)". However:

- this does not seem to be done much in practice;
- complex assertions increase run-time overhead, increasing the incentive to disable checking;
- a complex assertion might itself contain errors, leading to a false sense of confidence in the correctness of the program.

### 6.5.2 Repeated Inheritance

Listing 24 shows a collection of classes (Meyer 1992, page169). The features of `HomeBusiness` include:

- `address`, inherited once but by two paths;
- `residenceValue`, inherited from `value` in class `House`;
- `businessValue`, inherited from `value` in class `House`.

Repeated inheritance can be used to solve various problems that appear in OOP. For example, it is difficult in some languages to provide collections with more than one iteration sequence. In Eiffel, more than one sequence can be obtained by repeated inheritance of the successor function.

C++ has a similar mechanism, but it applies to entire classes (via the `virtual` mechanism) rather than to individual attributes of classes.

**Listing 25:** Exceptions in Eiffel

```
tryIt is
  local
    triedFirst:  Boolean -- initially false
  do
    if not triedFirst
      then firstmethod
      else secondmethod
    end
  rescue
    if not triedFirst
      then
        triedFirst := true
        retry
      else
        restoreInvariant
        -- report failure
    end
end
```

### 6.5.3 Exception Handling

An Eiffel function must either fulfill its contract or report failure. If a function contains a `rescue` clause, this clause is invoked if an operation within the function reports failure. A return from the `rescue` clause indicate that the function has failed. However, a `rescue` clause may perform some cleaning-up actions and then invoke `retry` to attempt the calculation again. The function in Listing 25 illustrates the idea. The mechanism seems harder to use than other, more conventional, exception handling mechanisms. It is not obvious that there are many circumstances in which it makes sense to "retry" a function.

**Exercise 33.** Early versions of Eiffel had a reference semantics (variable names denoted references to objects). After Eiffel had been used for a while, Meyer introduced ***expanded types***. An instance of an expanded type is an object that has a name (value semantics). Expanded types introduce various irregularities into the language, because they do not have the same properties as ordinary types. Why do you think expanded types were introduced? Was the decision to introduce them wise? □

## 6.6 Java

Java (Arnold and Gosling 1998) is an OOPL introduced by Sun Microsystems. Its syntax bears some relationship to that of C++, but Java is simpler in many ways than C++. Key features of Java include the following.

- Java is compiled to ***byte codes*** that are interpreted. Since any computer that has a Java byte code interpreter can execute Java programs, Java is highly portable.
- The ***portability*** of Java is exploited in network programming: Java bytes can be transmitted across a network and executed by any processor with an interpreter.

- ▷ Java offers ***security***. The byte codes are checked by the interpreter and have limited functionality. Consequently, Java byte codes do not have the potential to penetrate system security in the way that a binary executable (or even a MS-Word macro) can.
- ▷ Java has a class hierarchy with class `Object` at the root and provides ***single inheritance*** of classes.
- ▷ In addition to classes, Java provides ***interfaces*** with multiple inheritance.
- ▷ Java has an ***exception handling*** mechanism.
- ▷ Java provides ***concurrency*** in the form of threads.
- ▷ Primitive values, such as `int`, are not objects in Java. However, Java provides ***wrapper classes***, such as `Integer`, for each primitive type.
- ▷ A variable name in Java is a ***reference*** to an object.
- ▷ Java provides ***garbage collection***.

### 6.6.1 Portability

Compiling to byte codes is an implementation mechanism and, as such, is not strongly relevant to this course. The technique is quite old — Pascal P achieved portability in the same way during the mid-seventies — and has the disadvantage of inefficiency: Java programs are typically an order of magnitude slower than C++ programs with the same functionality. For some applications, such as simple Web “applets”, the inefficiency is not important. More efficient implementations are promised. One interesting technique is “just in time” compilation, in which program statements are compiled immediately before execution; parts of the program that are not needed during a particular run (this may be a significant fraction of a complex program) are not compiled.

The portability of Java has been a significant factor in the rapid spread of its popularity, particularly for Web programming.

### 6.6.2 Interfaces

A Java class, as usual in OOP, is a compile-time entity whose run-time instances are objects. An interface declaration is similar to a class declaration, but introduces only abstract methods and constants. Thus an interface has no instances.

A class may ***implement*** an interface by providing definitions for each of the methods declared in the interface. (The class may also make use of the values of the constants declared in the interface.)

A Java class can inherit from at most one parent class but it may inherit from several interfaces provided, of course, that it implements all of them. Consequently, the class hierarchy is a rooted tree but the interface hierarchy is a directed acyclic graph. Interfaces provide a way of describing and factoring the behaviour of classes.

Listing 26 shows a couple of simple examples of interfaces. The first interface ensures that instances of a class can be compared. (In fact, all Java classes inherit the function `equals` from class `Object` that can be redefined for comparison in a particular class.) If we want to store instances of a class in an ordered set, we need a way of sorting them as well: this requirement is specified by the second interface.

Class `Widget` is required to implement `equalTO` and `lessThan`:

```
class Widget implements Ordered
```

**Listing 26:** Interfaces

```
interface Comparable
{
  Boolean equalTo (Comparable other);
  // Return True if ‘this’ and ‘other’ objects are equal.
}
interface Ordered extends Comparable
{
  Boolean lessThan (Ordered other);
  // Return True if ‘this’ object precedes ‘other’ object in the ordering.
}
```

**Listing 27:** Declaring a Java exception class

```
public class DivideByZero extends Exception
{
  DivideByZero()
  {
    super("Division by zero");
  }
}
```

```
{
....
}
```

### 6.6.3 Exception Handling

The principal components of exception handling in Java are as follows:

- ▷ There is a class hierarchy rooted at class `Exception`. Instances of a subclass of class `Exception` are used to convey information about what has gone wrong. We will call such instances “exceptions”.
- ▷ The `throw` statement is used to signal the fact that an exception has occurred. The argument of `throw` is an exception.
- ▷ The `try` statement is used to attempt an operation and handle any exceptions that it raises.
- ▷ A `try` statement may use one or more `catch` statements to handle exceptions of various types.

The following simple example indicates the way in which exceptions might be used. Listing 27 shows the declaration of an exception for division by zero. The next step is to declare another exception for negative square roots, as shown in Listing 28

Next, we suppose that either of these problems may occur in a particular computation, as shown in Listing 29. Finally, we write some code that tries to perform the computation, as shown in Listing 30.

**Listing 28:** Another Java exception class

```
public class NegativeSquareRoot extends Exception
{
  public float value;
  NegativeSquareRoot(float badValue)
  {
    super("Square root with negative argument");
    value = badValue;
  }
}
```

**Listing 29:** Throwing exceptions

```
public void bigCalculation ()
  throws DivideByZero, NegativeSquareRoot
{
  ....
  if (....) throw new DivideByZero();
  ....
  if (....) throw new NegativeSquareRoot(x);
  ....
}
```

**Listing 30:** Using the exceptions

```
{
  try
  {
    bigCalculation()
  }
    catch (DivideByZero)
    {
        System.out.println("Oops!  divided by zero.");
    }
    catch (NegativeSquareRoot n)
    {
        System.out.println("Argument for square root was " + n);
    }
  }
}
```

### 6.6.4 Concurrency

Java provides concurrency in the form of ***threads***. Threads are defined by a class in that standard library. Java threads have been criticized on (at least) two grounds.

- ▷ The synchronization methods provided by Java (based on the early concepts of semaphores and monitors) are out of date and do not benefit from more recent research in this area.
- ▷ The concurrency mechanism is insecure. Brinch Hansen (1999) argues that Java's concurrency mechanisms are unsafe and that such mechanisms must be defined in the base language and checked by the compiler, not provided separately in a library.

**Exercise 34.** Byte codes provide portability. Can you suggest any other advantages of using byte codes? □

## 6.7 Kevo

All of the OOPLs previously described in this section are class-based languages. In a class-based language, the programmer defines one or more classes and, during execution, instances of these classes are created and become the objects of the system. (In Smalltalk, classes are run-time objects, but Smalltalk is nevertheless a class-based language.) There is another, smaller, family of OOPLs that use prototypes rather than classes.

Although there are several prototype-based OOPLs, we discuss only one of them here. Antero Taivalsaari (1993) has given a thorough account of the motivation and design of his language, Kevo. Taivalsaari (1993, page 172) points out that class-based OOPLs, starting with Simula, are based on an Aristotelian view in which

> the world is seen as a collection of objects with well-defined properties arranged [according] to a hierarchical taxonomy of concepts.

The problem with this approach is that there are many classes which people understand intuitively but which are not easily defined in taxonomic terms. Common examples include "book", "chair", "game", etc.

Prototypes provide an alternative to classes. In a prototype-based OOPL, each new type of object is introduced by defining a ***prototype*** or ***exemplar***, which is itself an object. Identical copies of a prototype — as many as needed — are created by ***cloning***. Alternatively, we can clone a prototype, modify it in some way, and use the resulting object as a prototype for another collection of identical objects.

The modified object need not be self-contained. Some of its methods will probably be the same as those of the original object, and these methods can be ***delegated*** to the original object rather than replicated in the new object. Delegation thus plays the role of inheritance in a prototype-based language and, for this reason, prototype-based languages are sometimes called ***delegation*** languages. This is misleading, however, because delegation is not the only possible mechanism for inheritance with prototypes.

**Example 16: Objects in Kevo.** This example illustrates how new objects are introduced in Kevo; it is based on (Taivalsaari 1993, pages 187–188). Listing 31 shows the introduction of a window object. The first line, `REF Window`, declares a new object with no properties. The second

**Listing 31:** Object Window

```
REF Window
Object.new → Window
Window ADDS
  VAR rect
  Prototypes.Rectangle.new → rect
  VAR contents
  METHOD drawFrame ....;
  METHOD drawContents ....;
  METHOD refresh
    self.drawFrame
    self.drawContents;
ENDADDS;
```

**Listing 32:** Object Title Window

```
REF TitleWindow
Window.new → TitleWindow
TitleWindow ADDS
  VAR title
  "Anonymous" → title
  METHOD drawFrame ....;
ENDADDS;
```

line defines `Window` as a clone of `Object`. In Kevo terminology, `Window` ***receives*** the basic printing and creation operations defined for `Object`. In the next block of code, properties are added to the window using the module operation `ADDS`.

The window has two components: `rect` and `contents`. The properties of `rect` are obtained from the prototype `Rectangle`. The properties of `contents` are left unspecified in this example. Finally, three new methods are added to `Window`. The first two are left undefined here, but the third, `refresh`, draws the window frame first and then the contents.

Listing 32 shows how to add a title to a window. It declares a new object, `TitleWindow` with `Window` as a prototype. Then it adds a new variable, `title`, with initial value `"Anonymous"`, and provides a new method `drawFrame` that overrides `Window.drawFrame` and draws a frame with a title. □

## 6.8 Other OOPLs

The OOPLs listed below are interesting, but lack of time precludes further discussion of them.

**Beta.** Beta is a descendant of Simula, developed in Denmark. It differs in many respects from "mainstream" OOPLs.

**Blue.** Blue (Kölling 1999) is a language and environment designed for teaching OOP. The language was influenced to some extent by Dee (Grogono 1991b). The environment is interesting in that users work directly with "objects" rather than program text and generated output.

**CLOS.** The Common LISP Object System is an extension to LISP that provides OOP. It has a number of interesting features, including "multi-methods" — the choice of a function is based on the class of all of its arguments rather than its first (possibly implicit) argument. CLOS also provides "before" and "after" methods — code defined in a superclass that is executed before or after the invocation of the function in a class.

**Self.** Self is the perhaps the best-known prototype language. Considerable effort has been put into efficient implementation, to the extent that Self programs run at up to half of the speed of equivalent C++ programs. This is an impressive achievement, given that Self is considerably more flexible than C++.

## 6.9 Evaluation of OOP

- ▷ OOP provides a way of building programs by ***incremental modification***.
- ▷ Programs can often be extended by adding new code rather than altering existing code.
- ▷ The mechanism for incremental modification without altering existing code is ***inheritance***. However, there are several different ways in which inheritance can be defined and used, and some of them conflict.
- ▷ The same identifier can be used in various contexts to name related functions. For example, all objects can respond to the message "print yourself".
- ▷ Class-based OOPLs can be designed so as to support information hiding (encapsulation) — the separation of interface and implementation. However, there may be conflicts between inheritance and encapsulation.
- ▷ An interesting, but rarely noted, feature of OOP is that the call graph is created and modified during execution. This is in contrast to procedural languages, in which the call graph can be statically inferred from the program text.

Section 12 contains a more detailed review of issues in OOP.

**Exercise 35.** Simula is 33 years old. Can you account for the slow acceptance of OOPLs by professional programmers? □

# 7 Backtracking Languages

The PLs that we discuss in this section differ in significant ways, but they have an important feature in common: they are designed to solve problems with multiple answers, and to find as many answers as required.

In mathematics, a many-valued expression can be represented as a relation or as a function returning a set. Consider the relation "divides", denoted by | in number theory. Any integer greater than 1 has at least two divisors (itself and 1) and possible others. For example, $1 \mid 6$, $2 \mid 6$, $3 \mid 6$, and $6 \mid 6$. The functional analog of the divides relation is the set-returning function `divisors`:

$$\begin{aligned} \texttt{divisors}(6) &= \{1, 2, 3, 6\} \\ \texttt{divisors}(7) &= \{1, 7\} \\ \texttt{divisors}(24) &= \{1, 2, 3, 4, 6, 8, 12, 24\} \end{aligned}$$

We could define functions that return sets in a FPL or we could design a language that works directly with relations. The approach that has been most successful, however, is to produce solutions one at a time. The program runs until it finds the first solution, reports that solution or uses it in some way, then backtracks to find another solution, reports that, and so on until all the choices have been explored or the user has had enough.

## 7.1 Prolog

The computational model (see Section 9.2) of a logic PL is some form of mathematical logic. Propositional calculus is too weak because it does not have variables. Predicate calculus is too strong because it is undecidable. (This means that there is no effective procedure that determines whether a given predicate is true.) Practical logic PLs are based on a restricted form of predicate calculus that has a decision procedure.

The first logic PL was Prolog, introduced by Colmerauer (1973) and Kowalski (1974). The discussion in this section is based on (Bratko 1990).

We can express ***facts*** in Prolog as predicates with constant arguments. All Prolog textbooks start with facts about someone's family: see Listing 33. We read `parent(pam, bob)` as "Pam is the parent of Bob". This Bob's parents are Pam and Tom; similarly for the other facts. Note that, since constants in Prolog start with a lower case letter, we cannot write `Pam`, `Bob`, etc.

Prolog is interactive; it prompts with `?-` to indicate that it is ready to accept a query. Prolog responds to simple queries about facts that it has been told with "`yes`" and "`no`", as in Listing 34. If the query contains a ***logical variable*** — an identifier that starts with an upper case letter — Prolog attempts to find a value of the variable that makes the query true.

**Listing 33:** A Family

```
parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```

**Listing 34:** Prolog queries

```
?- parent(tom, liz).
yes
?- parent(tom, jim).
no
```

If it succeeds, it replies with the binding.

```
?- parent(pam, X).
X = bob
```

If there is more than one binding that satisfies the query, Prolog will find them all. (The exact mechanism depends on the implementation. Some implementations require the user to ask for more results. Here we assume that the implementation returns all values without further prompting.)

```
?- parent(bob, C).
C = ann
C = pat
```

Consider the problem of finding Jim's grandparents. In logic, we can express "G is a grandparent of Jim" in the following form: "there is a person P such that P is a parent of Jim and there is a person G such that G is a parent of P". In Prolog, this becomes:

```
?- parent(P, jim), parent(G, P).
P = pat
G = bob
```

The comma in this query corresponds to conjunction ("and"): the responses to the query are bindings that make all of the terms valid simultaneously. Disjunction ("or") is provided by multiple facts or rules: in the example above, Bob is the parent of `C` is true if `C` is Ann ***or*** `C` is Pat.

We can expression the "grandparent" relation by writing a ***rule*** and adding it to the collection of facts (which becomes a collection of facts and rules). The rule for "grandparent" is "G is a grandparent of C if G is a parent of P and P is a parent of C". In Prolog:

```
grandparent(G, C) :- parent(G, P), parent(P, C).
```

Siblings are people who have the same parents. We could write this rule for siblings:

```
sibling(X, Y) :- parent(P, X), parent(P, Y).
```

Testing this rule reveals a problem.

```
?- sibling(pat, X).
X = ann
X = pat
```

We may not expect the second response, because we do not consider Pat to be a sibling of herself. But Prolog simply notes that the predicate `parent(P, X), parent(P, Y)` is satisfied by the bindings `P = bob`, `X = pat`, and `Y = pat`. To correct this rule, we would have to require that `X` and `Y` have different values:

```
sibling(X, Y) :- parent(P, X), parent(P, Y), different(X, Y).
```

It is clear that `different(X, Y)` must be built into Prolog somehow, because it would not be feasible to list every true instantiation of it as a fact.

$$\dfrac{\dfrac{\dfrac{\texttt{parent(pam, bob)}}{\texttt{ancestor(pam, bob)}} \qquad \texttt{parent(bob, pat)}}{\texttt{ancestor(pam, pat)}} \qquad \texttt{parent(pat, jim)}}{\texttt{ancestor(pam, jim)}}$$

Figure 4: Proof of `ancestor(pam, jim)`

Prolog allows rules to be ***recursive***: a predicate name that appears on the left side of a rule may also appear on the right side. For example, a person A is an ancestor of a person X if either A is a parent of X or A is an ancestor of a parent P of X:

```
ancestor(A, X) :- parent(A, X).
ancestor(A, X) :- ancestor(A, P), parent(P, X).
```

We confirm the definition by testing:

```
?- ancestor(pam, jim).
yes
```

We can read a Prolog program in two ways. Read ***declaratively***, `ancestor(pam, jim)` is a statement and Prolog's task is to prove that it is true. The proof is shown in Figure 4, which uses the convention that $\frac{P}{Q}$ means "from $P$ infer $Q$". Read ***procedurally***, `ancestor(pam, jim)` defines a sequence of fact-searches and rule-applications that establish (or fail to establish) the goal. The procedural steps for this example are as follows.

1. Using the first rule for `ancestor`, look in the data base for the fact `parent(pam, jim)`.

2. Since this search fails, try the second rule with `A = pam` and `X = jim`. This gives two new subgoals: `ancestor(pam, P)` and `parent(P, jim)`.

3. The second of these subgoals is satisfied by `P = pat` from the database. The goal is now `ancestor(pam, pat)`.

4. The goal expands to the subgoals `ancestor(pam, P)` and `parent(P, pat)`. The second of these subgoals is satisfied by `P = bob` from the database. The goal is now `ancestor(pam, bob)`.

5. The goal `ancestor(pam, bob)` is satisfied by applying the first rule with the known fact `parent(pam, bob)`.

In the declarative reading, Prolog finds multiple results simply because they are there. In the procedure reading, multiple results are obtained by ***backtracking***. Every point in the program at which there is more than one choice of a variable binding is called a ***choice point***. The choice points define a tree: the root of the tree is the starting point of the program (the main goal) and each leaf of the tree represents either ***success*** (the goal is satisfied with the chosen bindings) or ***failure*** (the goal cannot be satisfied with these bindings). Prolog performs a depth-first search of this tree.

Sometimes, we can tell from the program that backtracking will be useless. Declaratively, the predicate `minimum(X,Y,Min)` is satisfied if `Min` is the minimum of `X` and `Y`. We define rules for the cases $X \le Y$ and $X > Y$.

| Program | Meaning |
|---|---|
| `p :- a,b.  p :- c.` | $p \iff (a \wedge b) \vee c$ |
| `p :- a,!,b.  p:- c.` | $p \iff (a \wedge b) \vee (\sim a \wedge c)$ |
| `p :- c.  p :- a,!,b.` | $p \iff c \vee (a \wedge b)$ |

Figure 5: Effect of cuts

```
minimum(X, Y, X) :- X ≤ Y.
minimum(X, Y, Y) :- X > Y.
```

It is easy to see that, if one of these rules succeeds, we do not want to backtrack to the other, because it is certain to fail. We can tell Prolog not to backtrack by writing "`!`", pronounced "cut".

```
minimum(X, Y, X) :- X ≤ Y, !.
minimum(X, Y, Y) :- X > Y, !.
```

The introduction of the cut operation has an interesting consequence. If a Prolog program has no cuts, we can freely change the order of goals and clauses. These changes may affect the efficiency of the program, but they do not affect its declarative meaning. If the program has cuts, this is no longer true. Figure 5 shows an example of the use of cut and its effect on the meaning of a program.

Recall the predicate `different` introduced above to distinguish siblings, and consider the following declarations and queries.

```
different(X, X) :- !, fail.
different(X, Y).
?- different(Tom, Tom).
no
?- different(Ann, Tom).
yes
```

The query `different(Tom, Tom)` matches `different(X, X)` with the binding `X = Tom` and succeeds. The Prolog constant `fail` then reports failure. Since the cut has been passed, no backtracking occurs, and the query fails. The query `different(Ann, Tom)` does not match `different(X, X)` but it does match the second clause, `different(X, Y)` and therefore succeeds.

This idea can be used to define negation in Prolog:

```
not (P) :- P, !, fail.
true.
```

However, it is important to understand what negation in Prolog means. Consider Listing 35, assuming that the entire database is included. The program has stated correctly that crows fly and penguins do not fly. But it has made this inference only because the database has no information about penguins. The same program can produce an unlimited number of (apparently) correct responses, and also some incorrect responses. We account for this by saying that Prolog assumes a ***closed world***. Prolog can make inferences from the "closed world" consisting of the facts and rules in its database.

Now consider negation. In Prolog, proving `not(P)` means "`P` cannot be proved" which means "`P` cannot be inferred from the facts and rules in the database". Consequently, we should be sceptical about positive results obtained from Prolog programs that use `not`. The response

**Listing 35:** Who can fly?

```
        flies(albatross).
        flies(blackbird);
        flies(crow).
        ?- flies(crow).
        yes
        ?- flies(penguin).
        no
        ?- flies(elephant).
        no
        ?- flies(brick).
        no
        ?- flies(goose).
        no
```

```
        ?- not human(peter).
        yes
```

means no more than "the fact that Peter is human cannot be inferred from the database".

The Prolog system proceeds top-down. It attempts to match the entire goal to one of the rules and, having found a match, applies the same idea to the parts of the matched formulas. The "matching" process is called ***unification***.

Suppose that we have two terms $A$ and $B$ built up from constants and variables. A ***unifier*** of $A$ and $B$ is a substitution that makes the terms identical. A ***substitution*** is a binding of variable names to values. In the following example we follow the Prolog convention, using upper case letters $X, Y, \ldots$to denote variables and lower case letters $a, b, \ldots$to denote constants. Let $A$ and $B$ be the terms

$$\begin{aligned} A &\equiv \texttt{left}(\texttt{tree}(X, \texttt{tree}(a, Y)) \\ B &\equiv \texttt{left}(\texttt{tree}(b, Z)) \end{aligned}$$

and let

$$\sigma = (X = b, Z = \texttt{tree}(a, Y))$$

be a substitution. Then $\sigma A = \sigma B = \texttt{left}(\texttt{tree}(b, \texttt{tree}(a, Y))$ and $\sigma$ is a unifier of $A$ and $B$. Note that the unifier is not necessarily unique. However, we can define a ***most general unifier*** which is unique.

The unification algorithm contains a test called the ***occurs check*** that is used to reject a substitution of the form $X = f(X)$ in which a variable is bound to a formula that contains the variable. If we changed $A$ and $B$ above to

$$\begin{aligned} A &\equiv \texttt{left}(\texttt{tree}(X, \texttt{tree}(a, Y)) \\ B &\equiv \texttt{left}(\texttt{tree}(b, Y) \end{aligned}$$

they would not unify.

You can find out whether a particular version of Prolog implements the occurs check by trying the following query:

**Listing 36:** Restaurants

```
good(les_halles).
expensive(les_halles).
good(joes_diner).
reasonable(Restaurant) :  not expensive(Restaurant).
?- good(X), reasonable(X).
X = joes_diner.
?- reasonable(X), good(X).
no
```

```
?- X = f(X).
```

If the reply is “no”, the occurs check is implemented. If, as is more likely, the occurs check is not performed, the response will be

```
X = f(f(f(f( ....
```

Prolog syntax corresponds to a restricted form of first-order predicate calculus called ***clausal form logic***. It is not practical to use the full predicate calculus as a basis for a PL because it is undecidable. Clausal form logic is semi-decidable: there is an algorithm that will find a proof for any formula that is true in the logic. If a formula is false, the algorithm may fail after a finite time or may loop forever. The proof technique, called SLD resolution, was introduced by Robinson (1965). SLD stands for “***S***electing a literal, using a ***L***inear strategy, restricted to ***D***efinite clauses”.

The proof of validity of SLD resolution assumes that unification is implemented with the occurs check. Unfortunately, the occurs check is expensive to implement and most Prolog systems omit it. Consequently, most Prolog systems are technically unsound, although problems are rare in practice.

A Prolog program apparently has a straightforward interpretation as a statement in logic, but the interpretation is slightly misleading. For example, since Prolog works through the rules in the order in which they are written, the order is significant. Since the logical interpretation of a rule sequence is disjunction, it follows that disjunction in Prolog does not commute.

**Example 17: Failure of commutativity.** In the context of the rules

```
wife(jane).
female(X) :- wife(X).
female(X) :- female(X).
```

we obtain a valid answer to a query:

```
?- female(Y).
jane
```

But if we change the order of the rules, as in

```
wife(jane).
female(X) :- female(X).
female(X) :- wife(X).
```

the same query leads to endless applications of the second rule. □

**Exercise 36.** Listing 36 shows the results of two queries about restaurants. In logic, conjunction is commutative ($P \land Q \equiv Q \land P$). Explain why commutativity fails in this program. □

The important features of Prolog include:

- ▷ Prolog is based on a mathematical model (clausal form logic with SLD resolution).
- ▷ "Pure" Prolog programs, which fully respect the logical model, can be written but are often inefficient.
- ▷ In order to write efficient programs, a Prolog programmer must be able to read programs both declaratively (to check the logic) and procedurally (to ensure efficiency).
- ▷ Prolog implementations introduce various optimizations (the cut, omitting the occurs check) that improve the performance of programs but compromise the mathematical model.
- ▷ Prolog has garbage collection.

## 7.2 `Alma-0`

One of the themes of this course is the investigation of ways in which different paradigms of programming can be combined. The PL `Alma-0` takes as its starting point a simple imperative language, Modula–2 (Wirth 1982), and adds a number of features derived from logic programming to it. The result is a simple but highly expressive language. Some features of Modula–2 are omitted (the `CARDINAL` type; sets; variant records; open array parameters; procedure types; pointer types; the `CASE`, `WITH`, `LOOP`, and `EXIT` statements; nested procedures; modules). They are replaced by nine new features, described below. This section is based on Apt *et al.* (1998).

**BES.** Boolean expressions may be used as statements. The expression is evaluated. If its value is `TRUE`, the computation ***succeeds*** and the program continues as usual. If the value of the expression is `FALSE`, the computation ***fails***. However, this does not mean that the program stops: instead, control returns to the nearest preceding ***choice point*** and resumes with a new choice.

**Example 18: Ordering 1.** If the array `a` is ordered, the following statement succeeds. Otherwise, it fails; the loop exits, and `i` has the smallest value for which the condition is `FALSE`.

```
FOR i := 1 to n - 1 DO a[i] <= a[i+1] END
```

□

**SBE.** A statement sequence may be used as a boolean expression. If the computation of a sequence of statements succeeds, it is considered equivalent to a boolean expression with value `TRUE`. If the computation fails, it is considered equivalent to a boolean expression with value `FALSE`.

**Example 19: Ordering 2.** The following sequence decides whether `a` precedes `b` in the lexicographic ordering. The `FOR` statement finds the first position at which the arrays differ. If they are equal, the entire sequence fails. If there is a difference at position `i`, the sequence succeeds iff `a[i] < b[i]`.

```
NOT FOR i :=1 TO n DO a[i] = b[i] END;
a[i] < b[i]
```

□

**EITHER–ORELSE.** The construct `EITHER-ORELSE` is added to the language to permit backtracking. An `EITHER` statement has the form

```
EITHER <seq> ORELSE <seq> ....ORELSE <seq> END
```

**Listing 37:** Creating choice points

```
EITHER y := x
ORELSE x > 0; y := -x
END;
x := x + b;
y < 0
```

**Listing 38:** Pattern matching

```
SOME i := 0 TO n - m DO
  FOR j := 0 to m - 1 DO
    s[i+j] = p[j]
  END
END
```

Computation of an `EITHER` statement starts by evaluating the first sequence. If it succeeds, computation continues at the statement following `END`. If this computation subsequently fails, control returns to the second sequence in the `EITHER` statement, with the program in the same state as it was when the `EITHER` statement was entered. Each sequence is tried in turn in this way, until one succeeds or the last one fails.

**Example 20: Choice points.** Assume that the sequence shown in Listing 37 is entered with $x = a > 0$. The sequence of events is as follows:

- ▷ The assignment `y := x` assigns $a$ to `y`. This sequence succeeds and transfers control to `x := x + b`.
- ▷ The assignment `x := x + b` assigns $a + b$ to `x`.
- ▷ The test `y < 0` fails because $y = a > 0$.
- ▷ The original value of `x` is restored. The test `x > 0` succeeds because $x = a > 0$, and `y` receives the value $-a$.
- ▷ The final sequence is performed and succeeds, leaving $x = a + b$ and $y = -a$.

□

**SOME.** The `SOME` statement permits the construction of statements that work like `EITHER` statements but have a variable number of `ORELSE` clauses. The statement

```
SOME i := 1 TO n DO <seq> END
```

is equivalent to

```
EITHER i := 1; <seq>
ORELSE SOME i := 2 TO n DO <seq> END
```

**Example 21: Pattern matching.** Suppose that we have two arrays of characters: a pattern `p[0..m-1]` and a string `s[0..n-1]`. The sequence shown in Listing 38 sets `i` to the first occurrence of `p` in `s`, or fails if there is no match. □

**COMMIT.** The `COMMIT` statement provides a way of restricting backtracking. (It is similar to the cut in Prolog.) The statement

**Listing 39:** Checking the order

```
    COMMIT
      SOME i := 1 TO n DO
        a[i] <> b[i]
      END
    END;
    a[i] < b[i]
```

```
COMMIT <seq> END
```

evaluates the statement sequence `<seq>`; if `seq` succeeds (possibly by backtracking), all of the choice points created during its evaluation are removed.

**Example 22: Ordering 3.** Listing 39 shows another way of checking lexicographic ordering. With the `COMMIT`, this sequence yields that value of `a[i] < b[i]` for the smallest `i` such that `a[i] <> b[i]`. Without the `COMMIT`, it succeeds if `a[i] < b[i]` is `TRUE` for ***some*** value of `i` such that `a[i] <> b[i]`. □

**FORALL.** The `FORALL` statement provides for the exploration of all choice points generated by a statement sequence. The statement

```
FORALL <seq1> DO <seq2> END
```

evaluates `<seq1>` and then `<seq2>`. If `<seq1>` generates choice points, control backtracks through these, even if the original sequence succeeded. Whenever `<seq2>` succeeds, its choice points are removed (that is, there is an implicit `COMMIT` around `<seq2>`).

**Example 23: Matching.** Suppose we enclose the pattern matching code above in a function `Match`. The following sequence finds the number of times that `p` occurs in `s`.

```
count := 0;
FORALL k := Match(p, s) DO count := count + 1 END;
```

□

**EQ.** Variables in `Alma-0` start life as ***uninitialized*** values. After a value has been assigned to a variable, it ceases to be uninitialized and becomes ***initialized***. Expressions containing uninitialized values are allowed, but they do not have values. For example, the comparison `s = t` is evaluated as follows:

▷ if both `s` and `t` are initialized, their values are compared as usual.

▷ if one side is an uninitialized variable and the other is an expression with a value, this value is assigned to the variable;

▷ if both sides are uninitialized variables, an error is reported.

**Example 24: Remarkable sequences.**A sequence with 27 integer elements is ***remarkable*** if it contains three 1’s, three 2’s, . . . ., three 9’s arranged in such a way that for $i = 1, 2, \ldots, 9$ there are exactly $i$ numbers between successive occurrences of $i$. Here is a remarkable sequence:

$$(1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5, 7).$$

**Listing 40:** Remarkable sequences

```
TYPE Sequence = ARRAY [1..27] OF INTEGER;
PROCEDURE Remarkable (VAR a:  Sequence);
  VAR i, j:  INTEGER;
BEGIN
  For i := 1 TO 9 DO
    SOME j := 1 TO 25 - 2 * i DO
      a[j] = i;
      a[j+i+1] = i;
      a[j+2*i+1] = i
    END
  END
END;
```

**Listing 41:** Compiling a `MIX` parameter

```
VAR v:  integer;
BEGIN
  v := E;
  P(v)
END
```

**Problem 1.** Write a program that tests whether a given array is a remarkable sequence.

**Problem 2.** Find a remarkable sequence.

Due to the properties of "=", the program shown in Listing 40 is a solution for ***both*** of these problems! If the given sequence is initialized, the program tests whether it is remarkable; if it is not initialized, the program assigns a remarkable sequence to it. □

**MIX.** Modula–2 provides call by value (the default) and call by reference (indicated by writing `VAR` before the formal parameter. `Alma-0` adds a third parameter passing mechanism: ***call by mixed form***. The parameter is passed either by value or by reference, depending on the form of the actual parameter.

We can think of the implementation of mixed form in the following way. Suppose that procedure `P` has been declared as

```
PROCEDURE P (MIX n:  INTEGER); BEGIN ....END;
```

The call `P(v)`, in which `v` is a variable, is compiled in the usual way, interpreting `MIX` as `VAR`. The call `P(E)`, in which `E` is an expression, is compiled as if the programmer had written the code shown in Listing 41.

**Example 25: Linear Search.** The function `Find` in Listing 42 uses a mixed form parameter.

- ▷ The call `Find(7,a)` tests if 7 occurs in `a`.
- ▷ The call `Find(x, a)`, where `x` is an integer variable, assigns each component of `a` to `x` by backtracking.
- ▷ The sequence

```
Find(x, a); Find(x, b)
```

**Listing 42:** Linear search

```
Type Vector = ARRAY [1..N] of INTEGER;
PROCEDURE Find (MIX e:  INTEGER; a:  Vector);
  VAR i:  INTEGER;
BEGIN
  SOME i := 1 TO N DO e = a[i] END
END;
```

**Listing 43:** Generalized Addition

```
PROCEDURE Plus (MIX x, y, z:  INTEGER);
BEGIN
  IF KNOWN(x); KNOWN(y) THEN z = x + y
  ELSIF KNOWN(y); KNOWN(z) THEN x = z - y
  ELSIF KNOWN(x); KNOWN(z) THEN y = z - x
  END
END;
```

tests whether the arrays `a` and `b` have an element in common; if so, `x` receives the value of the common element.

▷ The statement

```
FORALL Find(x, a) DO Find(x, b) END
```

tests whether all elements of `a` are also elements of `b`.

▷ The statement

```
FORALL Find(x, a); Find(x, b) DO WRITELN(x) END
```

prints all of the elements that `a` and `b` have in common.

□

**KNOWN.** The test `KNOWN(x)` succeeds iff the variable `x` is initialized.

> **Example 26: Generalized Addition.** Listing 43 shows the declaration of a function `Plus`. The call `Plus(x,y,z)` will succeed if $x + y = z$ provided that at least two of the actual parameters have values. For example, `Plus(2,3,5)` succeeds and `Plus(5,n,12)` assigns 7 to `n`. □

The language `Alma-0` has a ***declarative semantics*** in which the constructs described above correspond to logical formulas. It also has a ***procedural semantics*** that describes how the constructs can be efficiently executed, using backtracking. `Alma-0` demonstrates that, if appropriate care is taken, it is possible to design PLs that smoothly combine paradigms without losing the advantages of either paradigm.

The interesting feature of `Alma-0` is that it demonstrates that it is possible to start with a simple, well-defined language (Modula-2), remove features that are not required for the current purpose, and add a small number of simple, new features that provide expressive power when used together.

**Exercise 37.** The new features of `Alma-0` are orthogonal, both to the existing features of Modula-2 and to each other. How does this use of orthogonality compare with the orthogonal design of Algol 68? □

## 7.3 Other Backtracking Languages

Prolog and `Alma-0` are by no means the only PLs that incorporate backtracking. Others include Icon (Griswold and Griswold 1983), SETL (Schwartz, Dewar, Dubinsky, and Schonberg 1986), and 2LP (McAloon and Tretkoff 1995).

***Constraint*** PLs are attracting increasing attention. A program is a set of constraints, expressed as equalities and inequalities, that must be satisfied. Prolog is strong in logical deduction but weak in numerical calculation; constraint PLs attempt to correct this imbalance.

# 8 Implementation

This course is concerned mainly with the nature of PLs. Implementation techniques are a separate, and very rich, topic. Nevertheless, there are few interesting things that we can say about implementation, and that is the purpose of this section.

## 8.1 Compiling and Interpreting

### 8.1.1 Compiling

As we have seen, the need for programming languages at a higher level than assembly language was recognized early. People also realized that the computer itself could perform the translation from a high level notation to machine code. In the following quote, "conversion" is used in roughly the sense that we would say "compilation" today.

> On the other hand, large storage capacities will lead in the course of time to very elaborate conversion routines. Programmes in the future may be written in a very different language from that in which the machines execute them. It is to be hoped (and it is indeed one of the objects of using a conversion routine) that many of the tiresome blunders that occur in present-day programmes will be avoided when programmes can be written in a language in which the programmer feels more at home. However, any mistake that the programmer does commit may prove more difficult to find because it will not be detected until the programme has been converted into machine language. Before he can find the cause of the trouble the programmer will either have to investigate the conversion process or enlist the aid of checking routines which will 'reconvert' and provide him with diagnostic information in his own language. (Gill 1954)

A ***compiler*** translates source language to machine language. Since practical compilation systems allow programs to be split into several components for compiling, there are usually several phases.

1. The components of the source program is compiled into object code.

2. The object code units and library functions required by the program are linked to form an executable file.

3. The executable file is loaded into memory and executed.

Since machine code is executed directly, compiled programs ideally run at the maximum possible speed of the target machine. In practice, the code generated by a compiler is usually not optimal. For very small programs, a programmer skilled in assembly language may be able to write more efficient code than a compiler can generate. For large programs, a compiler can perform global optimizations that are infeasible for a human programmer and, in any case, hand-coding such large programs would require excessive time.

All compilers perform some checks to ensure that the source code that they are translating is correct. In early languages, such as FORTRAN, COBOL, PL/I, and C, the checking was quite superficial and it was easy to write programs that compiled without errors but failed when they were executed. The value of compile-time checking began to be recognized in the late 60s, and languages such as Pascal, Ada, and C++ provided more thorough checking than their predecessors. The goal of checking is to ensure that, if a program compiles without errors, it will not fail when it is executed. This goal is hard to achieve and, even today, there are few languages that achieve it totally.

Compiled PLs can be classified according to whether the compiler can accept program components or only entire programs. An Algol 60 program is a block and must be compiled as a block. Standard Pascal programs cannot be split up, but recent dialects of Pascal provide program components.

The need to compile programs in parts was recognized at an early stage: even FORTRAN was implemented in this way. The danger of separate compilation is that there may be inconsistencies between the components that the compiler cannot detect. We can identify various levels of checking that have evolved.

- ▷ A FORTRAN compiler provides no checking between components. When the compiled components are linked, the linker will report subroutines that have not been defined but it will not detect discrepancies in the number of type of arguments. The result of such a discrepancy is, at best, run-time failure.
- ▷ C and C++ rely on the programmer to provide header files containing declarations and implementation files containing definitions. The compiler checks that declarations and definitions are compatible.
- ▷ Overloaded functions in C++ present a problem because one name may denote several functions. A key design decision in C++, however, was to use the standard linker (Stroustrup 1994, page 34). The solution is “name mangling” — the C++ compiler alters the names of functions by encoding their argument types in such a way that each function has a unique name.
- ▷ Modula-2 programs consist of interface modules and implementation modules. When the compiler compiles an implementation module, it reads all of the relevant interfaces and performs full checking. In practice, the interface modules are themselves compiled for efficiency, but this is not logically necessary.
- ▷ Eiffel programs consist of classes that are compiled separately. The compiler automatically derives an interface from the class declaration and uses the generated interfaces of other classes to perform full checking. Other OOPLs that use this method include Blue, Dee, and Java.

### 8.1.2 Interpreting

An ***interpreter*** accepts source language and run-time input data and executes the program directly. It is possible to interpret without performing any translation at all, but this approach tends to be inefficient. Consequently, most interpreters do a certain amount of translation before executing the code. For example, the source program might be converted into an efficient internal representation, such as an abstract syntax tree, which is then “executed”.

Interpreting is slower than executing compiled code, typically by a factor of 10 or more. Since no time is spent in compiling, however, interpreted languages can provide immediate response. For this reason, prototyping environments often provide an interpreted language. Early BASIC and LISP systems relied on interpretation, but more recent interactive systems provide an option to compile and may even provide compilation only.

Most interpreters check correctness of programs during execution, although some interpreters may report obvious (e.g., syntactic) errors as the program is read. Language such as LISP and Smalltalk are often said to be untyped, because programs do not contain type declarations, but this is incorrect. The type checking performed by both of these languages is actually more thorough than that performed by C and Pascal compilers; the difference is that errors are not detected until the offending statements are actually executed. For safety-critical applications, this may be too late.

### 8.1.3 Mixed Implementation Strategies

Some programming environments, particularly those for LISP, provide both interpretation and compilation. In fact, some LISP environments allow source code and compiled code to be freely mixed.

**Byte Codes.** Another mixed strategy consists of using a compiler to generate byte codes and an interpreter to execute the byte codes. The "byte codes", as the name implies, are simply streams of bytes that represent the program: a machine language for an "abstract machine". One of the first programming systems to be based on byte codes was the Pascal "P" compiler.

The advantage of byte code programs is that they can, in principle, be run on any machine. Thus byte codes provide a convenient way of implementing "portable" languages.

The Pascal "P" compiler provided a simple method for porting Pascal to a new kind of machine. The "Pascal P package" consisted of the P-code compiler written in P-code. A Pascal implementor had to perform the following steps:

1. Write a P-code interpreter. Since P-code is quite simple, this is not a very arduous job.
2. Use the P-code interpreter to interpret the P-code compiler. This step yields a P-code program that can be used to translate Pascal programs to P-code. At this stage, the implementor has a working Pascal system, but it is inefficient because it depends on the P-code interpreter.
3. Modify the compiler source so that the compiler generates machine code for the new machine. This is a somewhat tedious task, but it is simplified by the fact that the compiler is written in Pascal.
4. Use the compiler created in step 2 to compile the modified compiler. The result of this step is a P-code program that translates Pascal to machine code.
5. Use the result of step 4 to compile the modified compiler. The result is a machine code program that translates Pascal to machine code — in other words, a Pascal compiler for the new machine.

Needless to say, all of these steps must be carried out with great care. It is important to ensure that each step is completed correctly before beginning the next.

With the exercise of due caution, the process described above (called ***bootstrapping***) can be simplified. For example, the first version of the compiler, created in step 2 does not have to compile the entire Pascal language, but only those parts that are needed by the compiler itself. The complete sequence of steps can be used to generate a Pascal compiler that can compile itself, but may be incomplete in other respects. When that compiler is available, it can be extended to compile the full language.

Java uses byte codes to obtain portability in a different way. Java was designed to be used as an internet language. Any program that is to be distributed over the internet must be capable of running on all the servers and clients of the network. If the program is written in a conventional language, such as C, there must be a compiled version for every platform (a ***platform*** consists of a machine and operating system). If the program is written in Java, it is necessary only to provide a Java byte code interpreter for each platform.

**Just In Time.** "Just in time" (JIT) compilation is a recent strategy that is used for Java and a few research languages. The idea is that the program is interpreted, or perhaps compiled rapidly without optimization. As each statement is executed, efficient code is generated for it. This approach is based on the following assumptions:

- ▷ in a typical run, much of the code of a program will not be executed;
- ▷ if a statement has been executed once, it is likely to be executed again.

JIT provides fast response, because the program can be run immediately, and efficient execution, because the second and subsequent cycles of a loop are executed from efficiently compiled code. Thus JIT combines some of the advantages of interpretation and compilation.

**Interactive Systems** A PL can be designed for writing large programs that run autonomously or for interacting with the user. The categories overlap: PLs that can be used interactively, such as BASIC, LISP, SML, and Smalltalk, can also be used to develop large programs. The converse is not true: an interactive environment for C++ would not be very useful.

An interactive PL is usually interpreted, but interpretation is not the only possible means of implementation. PLs that start out with interpreters often acquire compilers at a later stage — BASIC, LISP, and a number of other interactive PLs followed this pattern of development.

A PL that is suitable for interactive use if programs can be described in small chunks. FPLs are often supported interactively because a programmer can build up a program as a collection of function definitions, where most functions are fairly small (e.g., they fit on a screen). Block structured languages, and modular languages, are less suitable for this kind of program development.

### 8.1.4 Consequences for Language Design

Any PL can be compiled or interpreted. A PL intended for interpretation, however, is usually implemented interactively. This implies that the user must be able to enter short sequences of code that can be executed immediately in the current context. Many FPLs are implemented in this way (often with a compiler); the user can define new functions and invoke any function that has previously been defined.

## 8.2 Garbage Collection

There is a sharp distinction: either a PL is implemented with garbage collection (GC) or it is not. The implementor does not really have a choice. An implementation of LISP, Smalltalk, or Java without GC would be useless — programs would exhaust memory in a few minutes at most. Conversely, it is almost impossible to implement C or C++ with GC, because programmers can perform arithmetic on pointers.

The imperative PL community has been consistently reluctant to incorporate GC: there is no widely-used imperative PL that provides GC. The functional PL community has been equally consistent, but in the opposite direction. The first FPL, LISP, provides GC, and so do all of its successors.

The OOP community is divided on the issue of GC, but most OOPLs provide it: Simula, Smalltalk, CLU, Eiffel, and Java. The exception, of course, is C++, for reasons that have been clearly explained (Stroustrup 1994, page 219).

> I deliberately designed C++ not to rely on [garbage collection]. I feared the very significant space and time overheads I had experienced with garbage collectors. I feared the added complexity of implementation and porting that a garbage collector would impose. Also, garbage collection would make C++ unsuitable for many of the low-level tasks for which it was intended. I like the idea of garbage collection as a mechanism that simplifies design and eliminates a source of errors. However, I am fully convinced that had garbage collection been an integral part of C++ originally, C++ would have been stillborn.

It is interesting to study Stroustrup's arguments in detail. In his view, the "fundamental reasons" that garbage collection is desirable are (Stroustrup 1994, page 220):

1. Garbage collection is easiest for the user. In particular, it simplifies the building and use of ***some*** libraries.

2. Garbage collection is more reliable than user-supplied memory management schemes for ***some*** applications.

The emphasis is added: Stroustrup makes it clear that he would have viewed GC in a different light if it provided advantages for ***all*** applications.

Stroustrup provides more arguments against GC than in favour of it (Stroustrup 1994, page 220):

1. Garbage collection causes run-time and space overheads that are not affordable for many current C++ applications running on current hardware.

2. Many garbage collection techniques imply service interruptions that are not acceptable for important classes of applications, such as hard real-time applications, device drivers, control applications, human interface code on slow hardware, and operating system kernels.

3. Some applications do not have the hardware resources of a traditional general-purpose computer.

4. Some garbage collection schemes require banning several basic C facilities such as pointer arithmetic, unchecked arrays, and unchecked function arguments as used by `printf()`.

5. Some garbage collection schemes impose constraints on object layout or object creation that complicates interfacing with other languages.

It is hard to argue with Stroustrup's conclusion (Stroustrup 1994, page 220):

> I know that there are more reasons for and against, but no further reasons are needed. These are sufficient arguments against the view that ***every*** application would be better done with garbage collection. Similarly, these are sufficient arguments against the view that ***no*** application would be better done with garbage collection. (Stroustrup 1994, page 220) [Emphasis in the original.]

Stroustrup's arguments do not preclude GC as an ***option*** for C++ and, in fact, Stroustrup advocates this. But it remains true that it is difficult to add GC to C++ and the best that can be done is probably "conservative" GC. In the meantime, most C++ programmers spend large amounts of time figuring out when to call destructors and debugging the consequences of their wrong decisions.

***Conservative*** GC depends on the fact that it is possible to guess with a reasonable degree of accuracy whether a particular machine word is a pointer (Boehm and Weiser 1988). In typical modern processors, a pointer is a 4-byte quantity, aligned on a word boundary, and its value is a legal address. By scanning memory and noting words with these properties, the GC can identify regions of memory that might be in use. The problem is that there will be occasionally be words that look like pointers but actually are not: this results in about 10% of garbage being retained. The converse, and more serious problem, of identifying live data as garbage cannot occur: hence the term "conservative".

The implementation of Eiffel provided by Meyer's own company, ISE, has a garbage collector that can be turned on and off. Eiffel, however, is a public-domain language that can be implemented by anyone who chooses to do so — there are in fact a small number of companies other than ISE that provide Eiffel. Implementors are encouraged, but not required to provide GC. Meyer (1992, page 335) recommends that an Eiffel garbage collector should have the following properties.

- ▷ It should be efficient, with a low overhead on execution time.
- ▷ It should be incremental, working in small, frequent bursts of activity rather than waiting for memory to fill up.
- ▷ It should be tunable. For example, programmers should be able to switch off GC in critical sections and to request a full clean-up when it is safe to do so.

The concern about the efficiency of GC is the result of the poor performance of early, mark-sweep collectors. Modern garbage collectors have much lower overheads and are "tunable" in Meyer's sense. Larose and Feeley (1999), for example, describe a garbage collector for a Scheme compiler. Programs written in Erlang, translated into Scheme, and compiled with this compiler are used in ATM switches. (Erlang is a proprietary functional language developed by Ericsson.)

GC has divided the programming community into two camps: those who use PLS with GC and consider it indispensable, and those who use PLs without GC and consider it unnecessary. Experience with C++ is beginning to affect the polarization, as the difficulties and dangers of explicit memory management in complex OO systems becomes apparent. Several commercial products that offer "conservative" GC are now available and their existence suggests that developers will increasingly respect the contribution of GC.

# 9 Abstraction

Abstraction is one of the most important mental tools in Computer Science. The biggest problems that we face in Computer Science are concerned with complexity; abstraction helps us to manage complexity. Abstraction is particularly important in the study of PLs because a PL is a two-fold abstraction.

A PL must provide an abstract view of the underlying machine. The values that we use in programming — integers, floats, booleans, arrays, and so on — are abstracted from the raw bits of computer memory. The control structures that we use — assignments, conditionals, loops, and so on — are abstracted from the basic machine instructions. The ***first task*** of a PL is to hide the low-level details of the machine and present a manageable interface, preferably an interface that does not depend on the particular machine that is being used.

A PL must also provide ways for the programmer to abstract information about the world. Most programs are connected to the world in one way or another, and some programs are little more than simulations of some aspect of the world. For example, an accounting program is, at some level, a simulation of tasks that used to be performed by clerks and accountants. The ***second task*** of a PL is to provide as much help as possible to the programmer who is creating a simplified model of the complex world in a computer program.

In the evolution of PLs, the first need came before the second. Early PLs emphasize abstraction from the machine: they provide assignments, loops, and arrays. Programs are seen as lists of instructions for the computer to execute. Later PLs emphasize abstraction from the world: they provide objects and processes. Programs are seen as descriptions of the world that happen to be executable.

The ***idea*** of abstracting from the world is not new, however. The design of Algol 60 began in the late fifties and, even at that time, the explicit goal of the designers was a “problem oriented language”.

**Example 27: Sets and Bits.** Here is an early example that contrasts the problem-oriented and machine-oriented approaches to PL design.

Pascal provides the problem-oriented abstraction `set`. We can declare types whose values are sets; we can define literal constants and variables of these types; and we can write expressions using set operations, as shown in Listing 44. In contrast, C provides bit-wise operations on integer data, as shown in Listing 45.

These features of Pascal and C have almost identical implementations. In each case, the elements

**Listing 44:** Sets in Pascal

```
type SmallNum = set of [1..10];
var s, t:  SmallNum;
var n:  integer;
begin
  s := [1, 3, 5, 7, 9];
  t := s + [2, 4, 6, 8];
  if (s <= t) ....
  if (n in s) ....
end
```

**Listing 45:** Bit operations in C

```
        int n = 0x10101010101;
        int mask = 0x10000;
        if (n & mask) ....
```

of a set are encoded as positions within a machine word; the presence or absence of an element is represented by a 1-bit or a 0-bit, respectively; and the operations for and-ing and or-ing bits included in most machine languages are used to manipulate the words.

The appearance of the program, however, is quite different. The Pascal programmer can think at the level of sets, whereas the C programmer is forced to think at the level of words and bits. Neither approach is necessarily better than the other: Pascal was designed for teaching introductory programming and C was designed for systems programmers. The trend, however, is away from machine abstractions such as bit strings and towards problem domain abstractions such as sets. □

## 9.1 Abstraction as a Technical Term

"Abstraction" is a word with quite general meaning. In Computer Science in general, and in PL in particular, it is often given a special sense. An ***abstraction mechanism*** provides a way of naming and parameterizing a program entity.

### 9.1.1 Procedures.

The earliest abstraction mechanism was the ***procedure***, which exists in rudimentary form even in assembly language. Procedural abstraction enables us to give a name to a statement, or group of statements, and to use the name elsewhere in the program to trigger the execution of the statements. Parameters enable us to customize the execution according to the particular requirements of the caller.

### 9.1.2 Functions.

The next abstraction mechanism, which also appeared at an early stage was the ***function***. Functional abstraction enables us to give a name to an expression, and to use the name to trigger evaluation of the expression.

In many PLs, procedures and functions are closely related. Typically, they have similar syntax and similar rules of construction. The reason for this is that an abstraction mechanism that works only for simple expressions is not very useful. In practice, we need to abstract form arbitrarily complex programs whose main purpose is to yield a single value.

The difference between procedures and functions is most evident at the call site, where a procedure call appears in a statement context and a function call appears in an expression context.

FORTRAN provides procedures (called "subroutines") and functions. Interestingly, COBOL does not provide procedure or function abstraction, although the `PERFORM` verb provides a rather restricted and unsafe way of executing statements remotely.

LISP provides function abstraction. The effect of procedures is obtained by writing functions with side-effects.

Algol 60, Algol 68, and C take the view that everything is an expression. (The idea seems to be that, since any computation leaves something in the accumulator, the programmer might as well be allowed to use it.) Thus statements have a value in these languages. It is not surprising that this viewpoint minimizes the difference between procedures and functions. In Algol 68 and C, a procedure is simple a function that returns a value of type `void`. Since this value is unique, it requires $\log 1 = 0$ bits of storage and can be optimized away by the compiler.

### 9.1.3 Data Types.

Although Algol 68 provided a kind of data type abstraction, Pascal was more systematic and more influential. Pascal provides primitive types, such as `integer`, `real`, and `boolean`, and type expressions (`C` denotes an integer constant and `T` denotes a type):

```
C..C
(C, C, ...., C)
array [T] of T
record v:  T, ....end
set of T
file of T
```

Type expressions can be named and the names can be used in variable declarations or as components of other types. Pascal, however, does not provide a parameter mechanism for types.

### 9.1.4 Classes.

Simula introduced an abstraction mechanism for classes. Although this was an important and profound step, with hindsight we can see that two simple ideas were needed: giving a name to an Algol block, and freeing the data component of the block from the tyranny of stack discipline.

## 9.2 Computational Models

There are various ways of describing a PL. We can write an informal description in natural language (e.g., the Pascal *User Manual and Report* (Jensen and Wirth 1976)), a formal description in natural language (e.g., the Algol report (Naur et al. 1960)), or a document in a formal notation (e.g., the Revised Report on Algol 68 (van Wijngaarden et al. 1975)). Alternatively, we can provide a formal semantics for the language, using the axiomatic, denotational, or operational approaches.

There is, however, an abstraction that is useful for describing a PL in general terms, without the details of every construct, and that is the ***computational model*** (also sometimes called the ***semantic model***). The computational model (CM) is an abstraction of the operational semantics of the PL and it describes the effects of various operations without describing the actual implementation.

Programmers who use a PL acquire intuitive knowledge of its CM. When a program behaves in an unexpected way, the implementation has failed to respect the CM.

**Example 28: Passing parameters.** Suppose that a PL specifies that arguments are passed by value. This means that the programs in Listing 46 and Listing 47 are equivalent. The statement $T\ a = E$ means “the variable $a$ has type $T$ and initial value $E$” and the statement $x \leftarrow a$ means “the value of $a$ is assigned (by copying) to $x$”.

**Listing 46:** Passing a parameter

```
void $f$ ($T\ x$);
{
   $S$;
}
$T\ a = E$;
$f(a)$;
```

**Listing 47:** Revealing the parameter passing mechanism

```
$T\ a = E$;
{
   $T\ x$;
   $x \leftarrow a$;
   $S$;
}
```

A compiler could implement this program by copying the value of the argument, as if executing the assignment $x \leftarrow a$. Alternatively, it could pass the address of the argument $a$ to the function but not allow the statement $S$ to change the value of $a$. Both implementations respect the CM. □

**Example 29: Parameters in FORTRAN.** If a FORTRAN subroutine changes the value of one of its formal parameters (called "dummy variables" in FORTRAN parlance) the value of the corresponding actual argument at the call site also changes. For example, if we define the subroutine

```
subroutine p (count)
integer count
....
count = count + 1
return
end
```

then, after executing the sequence

```
integer total
total = 0
call p(total)
```

the value of `total` is 1. The most common way of achieving this result is to pass the address of the argument to the subroutine. This enables the subroutine to both access the value of the argument and to alter it.

Unfortunately, most FORTRAN compilers use the same mechanism for passing constant arguments. The effect of executing

```
call p(0)
```

is to change all instances of 0 (the constant "zero") in the program to 1 (the constant "one")! The resulting bugs can be quite hard to detect for a programmer who is not familiar with FORTRAN's idiosyncrasies.

The programmer has a mental CM for FORTRAN which predicts that, although variable arguments may be changed by a subroutine, constant arguments will not be. The problem arises because typical FORTRAN compilers do not respect this CM. □

A simple CM does not imply easy implementation. Often, the opposite is true. For example, C is easy to compile, but has a rather complex CM.

**Example 30: Array Indexing.** In the CM of C, $a[i] \equiv *(a + i)$. To understand this, we must know that an array can be represented by a pointer to its first element and that we can access elements of the array by adding the index (multiplied by an appropriate but implicit factor) to this pointer.

In Pascal's CM, an array is a mapping. The declaration

```
var A :  array [1..N] of real;
```

introduces $A$ as a mapping:

$$A : \{ 1, \ldots, N \} \to \mathcal{R}.$$

A compiler can implement arrays in any suitable way that respects this mapping. □

**Example 31: Primitives in OOPLs.** There are two ways in which an OOPL can treat primitive entities such as booleans, characters, and integers.

- ▷ Primitive values can have a special status that distinguishes them from objects. A language that uses this policy is likely to be efficient, because the compiler can process primitives without the overhead of treating them as full-fledged objects. On the other hand, it may be confusing for the user to deal with two different kinds of variable — primitive values and objects.
- ▷ Primitive values can be given the same status as objects. This simplifies the programmer's task, because all variables behave like objects, but a naive implementation will probably be inefficient.

A solution for this dilemma is to define a CM for the language in which all variables are objects. An implementor is then free to implement primitive values efficiently provided that the behaviour predicted by the CM is obtained at all times.

In Smalltalk, every variable is an object. Early implementations of Smalltalk were inefficient, because all variables were actually implemented as objects. Java provides primitives with efficient implementations but, to support various OO features, also has classes for the primitive types. □

**Example 32: $\lambda$-Calculus as a Computational Model.** FPLs are based on the theory of partial recursive functions. They attempt to use this theory as a CM. The advantage of this point of view is that program text resembles mathematics and, to a certain extent, programs can be manipulated as mathematical objects.

The disadvantage is that mathematics and computation are not the same. Some operations that are intuitively simple, such as maintaining the balance of a bank account subject to deposits and withdrawals, are not easily described in classical mathematics. The underlying difficulty is that many computations depend on a concept of ***mutable state*** that does not exist in mathematics.

FPLs have an important property called ***referential transparency***. An expression is referentially transparent if its only attribute is its value. The practical consequence of referential transparency is that we can freely substitute equal expressions.

For example, if we are dealing with numbers, we can replace $E + E$ by $2 \times E$; the replacement would save time if the evaluation of $E$ requires a significant amount of work. However, if $E$ was not referentially transparent — perhaps because its evaluation requires reading from an input stream or the generation of random numbers — the replacement would change the meaning of the program. □

**Example 33: Literals in OOP.** How should an OOPL treat literal constants, such as 7? The following discussion is based on (Kölling 1999, pages 71–74).

There are two possible ways of interpreting the symbol "7": we could interpret it as an ***object constructor***, constructing the object 7; or as a constant reference to an object that already exists.

Dee (Grogono 1991a) uses a variant of the first alternative: literals are constructors. This introduces several problems. For example, what does the comparison `7 = 7` mean? Are there two 7-objects or just one? The CM of Dee says that the ***first*** occurrence of the literal "7" constructs a new object and ***subsequent*** occurrences return a reference to this (unique) object.

Smalltalk uses the second approach. Literals are constant references to objects. These objects, however, "just magically exist in the Smalltalk universe". Smalltalk does not explain where they come from, or when and how they are created.

Blue avoids these problems by introducing the new concept of a ***manifest class***. Manifest classes are defined by enumeration. The enumeration may be finite, as for class `Boolean` with members `true` and `false`. Infinite enumerations, as for class `Integer`, are allowed, although we cannot see the complete declaration.

The advantage of the Blue approach, according to Kölling (1999, page 72), is that:

> . . . . the distinction is made at the logical level. The object model explicitly recognizes these two different types of classes, and differences between numbers and complex objects can be understood ***independently of implementation concerns***. They cease to be anomalies or special cases at a technical level. This reduces the dangers of misunderstandings on the side of the programmer. [Emphasis added.]

□

**Exercise 38.** Do you think that Kölling's arguments for introducing a new concept (manifest classes) are justified? □

**Summary** Early CMs:

- ▷ modelled programs as code acting on data;
- ▷ structured programs by recursive decomposition of code.

Later CMs:

- ▷ modelled programs as packages of code and data;
- ▷ structured programs by recursive decomposition of packages of code and data.

The CM should:

- ▷ help programmers to reason about programs;
- ▷ help programmers to read and write programs;
- ▷ constrain but not determine the implementation.

# 10 Names and Binding

Names are a central feature of all programming languages. In the earliest programs, numbers were used for all purposes, including machine addresses. Replacing numbers by symbolic names was one of the first major improvements to program notation (Mutch and Gill 1954).

## 10.1 Free and Bound Names

The use of names in programming is very similar to the use of names in mathematics. We say that "$x$ occurs free" in an expression such as $x^2 + 2x + 5$. We do not know anything about the value of this expression because we do not know the value of $x$. On the other hand, we say that "$n$ occurs bound" in the expression

$$\sum_{n=0}^{5} (2n + 1). \tag{1}$$

More precisely, the ***binding occurrence*** of $n$ is $n = 0$ and the $n$ in the expression $(2n + 1)$ is a ***reference to***, or ***use of***, to this binding.

There are two things to note about (1). First, it has a definite value, 36, because $n$ is bound to a value (actually, the set of values $\{ 0, 1, \ldots, 5 \}$). Second, the particular name that we use does not change the value of the expression. If we systematically change $n$ to $k$, we obtain the expression $\sum_{k=0}^{5}(2k + 1)$, which has the same value as (1).

Similarly, the expression

$$\int_0^\infty e^{-x}\,dx,$$

which contains a binding occurence of $x$ (in $dx$) and a use of $x$ (in $e^{-x}$), has a value that does not depend on the particular name $x$.

The expression

$$\int e^{-x}\,dx,$$

is interesting because it binds $x$ but does not have a numerical value; the convention is that it denotes a function, $-e^{-x}$. The conventional notation for functions, however, is ambiguous. It is clear that $-e^{-x}$ is a function of $x$ because, by convention, $e$ denotes a constant. But what about $-e^{-ax}$: is it a function of $a$ or a function of $x$? There are several ways of resolving this ambiguity. We can write

$$x \mapsto -e^{-ax},$$

read "$x$ maps to $-e^{-ax}$". Or we can use the notation of the $\lambda$-calculus:

$$\lambda x \,.\, -e^{-ax}.$$

In predicate calculus, predicates may contain free names, as in:

$$n \bmod 2 = 0 \Rightarrow n^2 \bmod 2 = 0. \tag{2}$$

Names are bound by the quantifiers $\forall$ (for all) and $\exists$ (there exists). We say that the formula

$$\forall n \,.\, n \bmod 2 = 0 \Rightarrow n^2 \bmod 2 = 0$$

is ***closed*** because it contains no free variables. In this case, the formula is also true because (2) is true for all values of $n$. Strictly, we should specify the range of values that $n$ is allowed to assume. We could do this implicitly: for example, it is very likely that (2) refers to integers. Alternatively, we could define the range of values explicitly, as in

$$\forall n \in \mathcal{Z} \,.\, n \bmod 2 = 0 \Rightarrow n^2 \bmod 2 = 0.$$

Precisely analogous situations occur in programming. In the function

```
int f(int n)
{
  return k + n;
}
```

$k$ occurs free and $n$ occurs bound. Note that we cannot tell the value of $n$ from the definition of the function but we know that $n$ will be given a value when the function is called.

In most PLs, a program with free variables will not compile. A C compiler, for example, will accept the function $f$ defined above only if it is compiled in a scope that contains a declaration of $k$. Some early PLs, such as FORTRAN and PL/I, provided implicit declarations for variables that the programmer did not declare; this is now understood to be error-prone and is not a feature of recent PLs.

## 10.2 Attributes

In mathematics, a variable normally has only one attribute: its value. In (1), $n$ is bound to the values $0, 1, \ldots, 5$. Sometimes, we specify the domain from which these values are chosen, as in $n \in \mathcal{Z}$.

In programming, a name may have several attributes, and they may be bound at different times. For example, in the sequence

```
int n;
n = 6;
```

the first line binds the type `int` to $n$ and the second line binds the value 6 to $n$. The first binding occurs when the program is compiled. (The compiler records the fact that the type of $n$ is `int`; neither this fact nor the name $n$ appears explicitly in the compiled code.) The second binding occurs when the program is executed.

This example shows that there are two aspects of binding that we must consider in PLs: the attribute that is bound, and the time at which the binding occurs. The attributes that may be bound to a name include: type, address, and value. The times at which the bindings occur include: compile time, link time, load time, block entry time, and statement execution time.

**Definition.** A binding is ***static*** if it occurs during before the program is executed: during compilation or linking. A binding is ***dynamic*** if it occurs while the program is running: during loading, block entry, or statement execution.

**Example 34: Binding.** Consider the program shown in Listing 48. The address of $k$ is bound when the program starts; its value is bound by the `scanf` call. The address and value of $n$ are bound only when the function $f$ is called; if $f$ is not called (because $k \le 0$), then $n$ is never bound. □

**Listing 48:** Binding

```
void f()
{
  int n=7;
  printf("%d", n);
}
void main ()
{
  int k;
  scanf("%d", &k);
  if (k>0)
    f();
}
```

## 10.3 Early and Late Binding

An anonymous contributor to the *Encyclopedia of Computer Science* wrote:

> Broadly speaking, the history of software development is the history of ever-later binding time.

As a rough rule of thumb:

$$\begin{array}{rcl} \text{Early binding} & \Rightarrow & \text{efficiency;} \\ \text{Late binding} & \Rightarrow & \text{flexibility.} \end{array}$$

**Example 35: Variable Addressing.** In FORTRAN, addresses are bound to variable names at compile time. The result is that, in the compiled code, variables are addressed directly, without any indexing or other address calculations. (In reality, the process is somewhat more complicated. The compiler assigns an address relative to a compilation unit. When the program is linked, the address of the unit within the program is added to this address. When the program is loaded, the address of the program is added to the address. The important point is that, by the time execution begins, the "absolute" address of the variable is known.)

FORTRAN is efficient, because absolute addressing is used. It is inflexible, because all addresses are assigned at load time. This leads to wasted space, because all local variables occupy space whether or not they are being used, and also prevents the use of direct or indirect recursion.

In languages of the Algol family (Algol 60, Algol 68, C, Pascal, etc.), local variables are allocated on the run-time stack. When a function is called, a stack frame (called, in this context, an ***activation record*** or AR) is created for it and space for its parameters and local variables is allocated in the AR. The variable must be addressed by adding an offset (computed by the compiler) to the address of the AR, which is not known until the function is called.

Algol-style is slightly less efficient than FORTRAN because addresses must be allocated at block-entry time and indexed addressing is required. It is more flexible than FORTRAN because inactive functions do not use up space and recursion works.

Languages that provide dynamic allocation, such as Pascal and C and most of their successors, have yet another way of binding an address to a variable. In these languages, a statement such as `new`$(p)$

allocates space on the "heap" and stores the address of that space in the pointer *p*. Accessing the variable requires indirect addressing, which is slightly slower then indexed addressing, but greater flexibility is obtained because dynamic variables do not have to obey stack discipline (last in, first out). □

**Example 36: Function Addressing.** When the compiler encounters a function definition, it binds an address to the function name. (As above, several steps must be completed before the absolute address of the function is determined, but this is not relevant to the present discussion.) When the compiler encounters a function invocation, it generates a call to the address that it has assigned to the function.

This statement is true of all PLs developed before OOP was introduced. OOPLs provide "virtual" functions. A virtual function name may refer to several functions. The actual function referred to in a particular invocation is not in general known until the call is executed.

Thus in non-OO PLs, functions are statically bound, with the result that calls are efficiently executed but no decisions can be made at run-time. In OOPLs, functions are dynamically bound. Calls take longer to execute, but there is greater flexibility because the decision as to which function to execute is made at run-time.

The overhead of a dynamically-bound function call depends on the language. In Smalltalk, the overhead can be significant, because the CM requires a search. In practice, the search can be avoided in up to 95% of calls by using a cache. In C++, the compiler generates "virtual function tables" and dynamic binding requires only indirect addressing. Note, however, that this provides yet another example of the principle: Smalltalk binds later than C++, runs more slowly, but provides greater flexibility. □

## 10.4 What Can Be Named?

The question "what can be named?" and its converse "what can be denoted?" are important questions to ask about a PL.

**Definition.** An entity can be ***named*** if the PL provides a definitional mechanism that associates a name with an instance of the entity. An entity can be ***denoted*** if the PL provides ways of expressing instances of the entity as expressions.

**Example 37: Functions.** In C, we can name functions but we cannot denote them. The definition

```
int f(int x)
{
  B
}
```

introduces a function with name $f$, parameter $x$, and body $B$. Thus functions in C can be named. There is no way that we can write a function without a name in C. Thus functions in C cannot be denoted.

The corresponding definition in Scheme would be:

```
(defun f (lambda (x)
  E))
```

This definition can be split into two parts: the name of the function being defined (`f`) and the function itself (`(lambda (x) E)`). The notation is analogous to that of the $\lambda$-calculus:

$$f \quad = \quad \lambda x \,.\, E$$

FPLs allow expressions analogous to $\lambda x \,.\, E$ to be used as functions. LISP, as mentioned in Section 5.1, is ***not*** one of these languages. However, Scheme is, and so are SML and other FPLs. □

**Example 38: Types.** In early versions of C, types could be denoted but not named. For example, we could write

```
        int *p[10];
```

but we could not give a name to the type of `p` ("array of 10 pointers to `int`"). Later versions of C introduced the `typedef` statement, making types nameable:

```
        typedef int *API[10];
        API p;
```

□

## 10.5 What is a Variable Name?

We have seen that most PLs choose between two interpretations of a variable name. In a PL with value semantics, variable names denote memory locations. Assigning to a variable changes the contents of the location named by the variable. In a PL with reference semantics, variable names denote objects in memory. Assigning to a variable, if permitted by the language, changes the denotation of the name to a different object.

Reference semantics is sometimes called "pointer semantics". This is reasonable in the sense that the implementation of reference semantics requires the storage of addresses — that is, pointers. It is misleading in that providing pointers is not the same as providing reference semantics. The distinction is particularly clear in Hoare's work on PLs. Hoare (1974) has this to say about the introduction of pointers into PLs.

> Many language designers have preferred to extend [minor, localized faults in Algol 60 and other PLs] throughout the whole language by introducing the concept of reference, pointer, or indirect address into the language as an assignable item of data. This immediately gives rise in a high-level language to one of the most notorious confusions of machine code, namely that between an address and its contents. Some languages attempt to solve this by even more confusing automatic coercion rules. Worse still, an indirect assignment through a pointer, just as in machine code, can update any store location whatsoever, and the damage is no longer confined to the variable explicitly names as the target of assignment. For example, in Algol 68, the assignment
>
> ```
>         x := y
> ```
>
> always changes `x`, but the assignment
>
> ```
>         p := y + 1
> ```
>
> may, if `p` is a reference variable, change any other variable (of appropriate type) in the whole machine. One variable it can ***never*** change is `p`! . . . . References are like jumps,

> leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover.

One year later, Hoare (1975) provided his solution to the problem of references in high-level PLs:

> In this paper, we will consider a class of data structures for which the amount of storage can actually vary during the lifetime of the data, and we will show that it can be satisfactorily accommodated in a high-level language using solely high-level problem-oriented concepts, and without the introduction of references.

The implementation that Hoare proposes in this paper is a reference semantics with types. Explicit types make it possible to achieve

> . . . . a significant improvement on the efficiency of compiled LISP, perhaps even a factor of two in space-time cost for suitable applications. (Hoare 1975)

All FPLs and most OOPLs (the notable exception, of course, being C++) use reference semantics. There are good reasons for this choice, but the reasons are not the same for each paradigm.

- In a FPL, all (or at least most) values are immutable. If $X$ and $Y$ are have the same value, the program cannot tell whether $X$ and $Y$ are distinct objects that happen to be equal, or pointers to the same object. It follows that value semantics, which requires copying, would be wasteful because there is no point in making copies of immutable objects.

  (LISP provides two tests for equality. `(eq x y)` is `true` if `x` and `y` are pointers to the same object. `(equal x y)` is `true` if the objects `x` and `y` have the same extensional value. These two functions are provided partly for efficiency and partly to cover up semantic deficiencies of the implementation. Some other languages provide similar choices for comparison.)
- One of the important aspects of OOP is ***object identity***. If a program object $X$ corresponds to a unique entity in the world, such as a person, it should be unique in the program too. This is most naturally achieved with a reference semantics.

The use of reference semantics in OOP is discussed at greater length in (Grogono and Chalin 1994).

## 10.6 Polymorphism

The word "polymorphism" is derived from Greek and means, literally, "many shapes". In PLs, "polymorphism" is used to describe a situation in which one name can refer to several different entities. The most common application is to functions. There are several kinds of polymorphism; the terms "ad hoc polymorphism" and "parametric polymorphism" are due to Christopher Strachey.

### 10.6.1 Ad Hoc Polymorphism

In the code

```
int m,n;
float x,y;
printf("%d %f", m+n, x+y);
```

the symbol "+" is used in two different ways. In the expression $m + n$, it stands for the function that adds two integers. In the expression $x + y$, it stands for the function that adds two floats.

In general, ad hoc polymorphism refers to the use of a single function name to refer to two or more distinct functions. Typically the compiler uses the types of the arguments of the function to decide which function to call. Ad hoc polymorphism is also called "overloading".

Almost all PLs provide ad hoc polymorphism for built-in operators such as "+", "−", "*", etc.

Ada, C++, and other recent languages also allow programmers to overload functions. (Strictly, we should say "overload function names" but the usage "overloaded functions" is common.) In general, all that the programmer has to do is write several definitions, using the same function name, but ensuring that the either the number or the type of the arguments are different.

**Example 39: Overloaded Functions.** The following code declares three distinct functions in C++.

```
int max (int x, int y);
int max (int x, int y, int z);
float max (float x, float y);
```

□

### 10.6.2 Parametric Polymorphism

Suppose that a language provides the type `list` as a parameterized type. That is, we can make declarations such as these:

```
list(int)
list(float)
```

Suppose also that we have two functions: `sum` computes the sum of the components of a given list, and `len` computes the number of components in a given list. There is an important difference between these two functions. In order to compute the sum of a list, we must be able to choose an appropriate "add" function, and this implies that we must know the type of the components. On the other hand, there seems to be no need to know the type of the components if all we need to do is count them.

A function such as `len`, which counts the components of a list but does not care about their type, has ***parametric polymorphism***.

**Example 40: Lists in SML.** In SML, functions can be defined by cases, "[]" denotes the empty list, and "::" denotes list construction. We write definitions without type declarations and SML infers the types. Here are functions that sum the components of a list and count the components of a list, respectively.:

```
sum [] = 0
| sum (x::xs) = x + sum(xs)
len [] = 0
| len(x::xs) = 1 + len(xs)
```

SML infers the type of `sum` to be `list(int)` → `int`: since the sum of the empty list is (integer) 0, the type of the operator "+" must be `int` × `int` → `int`, and all of the components must be integers.

For the function `len`, SML can infer nothing about the type of the components from the function definition, and it assigns the type `list`$(\alpha) \rightarrow$ `int` to the function. $\alpha$ is a type variable, acting as a type parameter. When `len` is applied to an actual list, $\alpha$ will implicitly assume the type of the list components. □

### 10.6.3 Object Polymorphism

In OOPLs, there may be many different objects that provide a function called, say, $f$. However, the effect of invoking $f$ may depend on the object. The details of this kind of polymorphism, which are discussed in Section 6, depend on the particular OOPL. This kind of polymorphism is a fundamental, and very important, aspect of OOP.

## 10.7 Assignment

Consider the assignment $x := E$. Whatever the PL, the semantics of this statement will be something like:

- ▷ evaluate the expression $E$;
- ▷ store the value obtained in $x$.

The assignment is unproblematic if $x$ and $E$ have the same type. But what happens if they have different types? There are several possibilities.

- ▷ The statement is considered to be an error. This will occur in a PL that provides static type checking but does not provide coercion. For example, Pascal provides only a few coercions (subrange to `integer`, `integer` to `real`, etc.) and rejects other type differences.
- ▷ The compiler will generate code to convert the value of expression $E$ to the type of $x$. This approach was taken to extremes in COBOL and PL/I. It exists in C and C++, but C++ is stricter than C.
- ▷ The value of $E$ will be assigned to $x$ anyway. This is the approach taken by PLs that use dynamic type checking. Types are associated with objects rather than with names.

## 10.8 Scope and Extent

The two most important properties of a variable name are ***scope*** and ***extent***.

### 10.8.1 Scope

The ***scope*** of a name is the region of the program text in which a name may be used. In C++, for example, the scope of a local variable starts at the declaration of the variable and ends at the end of the block in which the declaration appears, as shown in Listing 49. Scope is a ***static*** property of a name that is determined by the semantics of the PL and the text of the program. Examples of other scope rules follow.

**FORTRAN.** Local variables are declared at the start of subroutines and at the start of the main program. Their scope starts at the declarations and end at the end of the program unit (that

**Listing 49:** Local scope in C++

```
{
  // x not accessible.
  t x;
  // x accessible
  ....
  {
     // x still accessible in inner block
    ....
  }
   // x still accessible
}
// x not accessible
```

**Listing 50:** FORTRAN `COMMON` blocks

```
subroutine first
integer i
real a(100)
integer b(250)
common /myvars/ a, b
....
return
end
subroutine second
real x(100), y(100)
integer k(250)
common /myvars/ x, k, y
....
return
end
```

is, subroutine or main program) within which the declaration appears. Subroutine names have global scope. It is also possible to make variables accessible in selected subroutines by means of `COMMON` statements, as shown in Listing 50. This method of "scope control" is insecure because the compiler does not attempt to check consistency between program units.

**Algol 60.** Local variables are declared at the beginning of a block. Scopes are ***nested***: a declaration of a name in an inner block hides a declaration of the same name in an outer block, as shown in Listing 51.

The fine control of scope provided by Algol 60 was retained in Algol 68 but simplified in other PLs of the same generation.

**Pascal.** Control structures and nested functions are nested but declarations are allowed only at the start of the program or the start of a function declaration. In early Pascal, the order of declarations was fixed: constants, types, variables, and functions were declared in that order. Later dialects of Pascal sensibly relaxed this ordering.

Pascal also has a rather curious scoping feature: within the statement `S` of the statement

**Listing 51:** Nested scopes in Algol 60

```
begin
  real x; integer k;
  begin
    real x;
    x := 3.0
  end;
  x := 6.0;
end
```

`with r do S`, a field `f` of the record `r` may be written as `f` rather than as `r.f`.

**C.** Function declarations cannot be nested and all local variables must appear before executable statements in a block. The complete rules of C scoping, however, are complicated by the fact that C has five ***overloading classes*** (Harbison and Steele 1987):

- ▷ preprocessor macro names;
- ▷ statement labels;
- ▷ structure, union, and enumeration tags;
- ▷ component names;
- ▷ other names.

Consequently, the following declarations do not conflict:

```
{
  char str[200];
  struct str {int m, n; } x;
  ....
}
```

**Inheritance.** In OOPLs, inheritance is, in part, a scoping mechanism. When a child class `C` inherits a parent class `P`, some or all of the attributes of `P` are made visible in `C`.

**C++.** There are a number of additional scope control mechanisms in C++. Class attributes may be declared `private` (visible only with the class), `protected` (visible within the class and derived classes), and `public` (visible outside the class). Similarly, a class may be derived as `private`, `protected`, or `public`. A function or a class may be declared as a `friend` of a class, giving it access to the private attributes of the class. Finally, C++ follows C in providing directives such as `static` and `extern` to modify scope.

**Global Scope.** The name is visible throughout the program. Global scope is useful for pervasive entities, such as library functions and fundamental constants ($\pi = 3.1415926$) but is best avoided for application variables.

FORTRAN does not have global variables, although programmers simulate them by over using `COMMON` declarations. Subroutine names in FORTRAN are global.

Names declared at the beginning of the outermost block of Algol 60 and Pascal programs have global scope. (There may be "holes" in these global scopes if the program contains local declarations of the same names.)

**Listing 52:** Declaration before use

```
void main ()
{
  const int i = 6;
  {
    int j = i;
    const int i = 7;
    cout << j << endl;
  }
}
```

**Listing 53:** Qualified names

```
class Widget
{
public:
  void f();
};
Widget w;
f(); // Error:  f is not in scope
w.f(); // OK
```

The largest default scope in C is “file scope”. Global scope can be obtained using the storage class `extern`.

**Local Scope.** In block-structured languages, names declared in a block are local to the block. There is a question as to whether the scope starts at the point of definition or is the entire block. (In other words, does the PL require “declaration before use”?) C++ uses the “declare before use” rule and the program in Listing 52 prints 6.

In Algol 60 and C++, local scopes can be as small as the programmer needs. Any statement context can be instantiated by a block that contains local variable declarations. In other languages, such as Pascal, local declarations can be used only in certain places. Although a few people do not like the fine control offered by Algol 60 and C++, it seems best to provide the programmer with as much freedom as possible and to keep scopes as small as possible.

**Qualified Scope.** The components of a structure, such as a Pascal record or a C `struct`, have names. These names are usually hidden, but can be made visible by the name of the object. Listing 53 provides an example in C++. The construct `w.` opens a scope in which the public attributes of the object `w` are visible.

The dot notation also appears in many OOPLs, where is it used to select attributes and functions of an object. For an attribute, the syntax is the same: `o.a` denotes attribute `a` of object `o`. If the attribute is a function, a parameter list follows: `o.f(x)` denotes the invocation of function `f` with argument `x` in object `o`.

**Import and Export.** In modular languages, such as Modula-2, a name or group of names can be brought into a scope with an `import` clause. The module that provides the definitions must have a corresponding `export` clause.

Typically, if module `A` exports name `X` and module `B` imports name `X` from `A`, then `X` may be used in module `B` as if it was declared there.

**Namespaces.** The mechanisms above are inadequate for very large programs developed by large teams. Problems arise when a project uses several libraries that may have conflicting names. Namespaces, provided by PLs such as C++ and Common LISP, provide a higher level of name management than the regular language features.

In summary:

- ▷ Nested scopes are convenient in small regions, such as functions. The advantage of nested scopes for large structures, such as classes or modules, is doubtful.
- ▷ Nesting is a form of scope management. For large structures, explicit control by name qualification may be better than nesting.
- ▷ Qualified names work well at the level of classes and modules, when the source of names is obvious.
- ▷ The import mechanism has the problem that the source of a name is not obvious where it appears: users must scan possibly remote import declarations.
- ▷ Qualified names are inadequate for very large programs.
- ▷ Library designers can reduce the potential for name conflicts by using distinctive prefixes. For example, all names supplied by the commercial graphics library FastGraph are have `fg_` as a prefix.
- ▷ Namespaces provide a better solution than prefixes.

Scope management is important because the programmer has no "work arounds" if the scoping mechanisms provided by the PL are inadequate. This is because PLs typically hide the scoping mechanisms.

If a program is compiled, names will normally not be accessible at run-time unless they are included for a specific purpose such as debugging. In fact, one way to view compiling is as a process that converts names to numbers.

Interpreters generally have a repository that contains the value of each name currently accessible to the program, but programmers may have limited access to this repository.

### 10.8.2 Are nested scopes useful?

As scope control mechanisms became more complicated, some people began to question to need for nested scopes (Clarke, Wilden, and Wolf 1980; Hanson 1981).

- ▷ On a small scale, for example in statement and functions, nesting works fairly well. It is, however, the cause of occasional errors that could be eliminated by requiring all names in a statement or function to be unique.
- ▷ On a large scale, for example in classes and modules, nesting is less effective because there is more text and a larger number of names. It is probably better to provide special mechanisms to provide the scope control that is needed rather than to rely on simple nesting.

C++ allows classes to be nested (this seems inconsistent, given that functions cannot be nested). The effect is that the nested class can be accessed only by the enclosing class. An alternative mechanism would be a declaration that states explicitly that class `A` can be accessed only by class `B`. This mechanism would have the advantage that it would also be able to describe constraints such as "class `A` can be accessed only by classes `B`, `C`, and `D`".

### 10.8.3 Extent

The ***extent***, also called ***lifetime***, of a name is the period of time during program execution during which the object corresponding to the name exists. Understanding the relation between scope and extent is an important part of understanding a PL.

Global names usually exist for the entire lifetime of the execution: they have ***global extent***.

In FORTRAN, local variables also exist for the lifetime of the execution. Programmers assume that, on entry to a subroutine, its local variables will not have changed since the last invocation of the subroutine.

In Algol 60 and subsequent stack-based languages, local variables are instantiated whenever control enters a block and they are destroyed (at least in principle) when control leaves the block. It is an error to create an object on the stack and to pass a reference to that object to an enclosing scope. Some PLs attempt to detect this error (e.g., Algol 68), some try to prevent it from occurring (e.g., Pascal), and others leave it as a problem for the programmer (e.g., C and C++).

In PLs that use a reference model, objects usually have unlimited extent, whether or not their original names are accessible. Examples include Simula, Smalltalk, Eiffel, CLU, and all FPLs. The advantage of the reference model is that the problems of disappearing objects (dangling pointers) and inaccessible objects (memory leaks) do not occur. The disadvantage is that GC and the accompanying overhead are more or less indispensable.

The separation of extent from scope was a key step in the evolution of post-Algol PLs.

# 11 Structure

The earliest programs had little structure. Although a FORTRAN program has a structure — it consists of a main program and subroutines — the main program and the subroutines themselves are simply lists of statements.

## 11.1 Block Structure

Algol 60 was the first major PL to provide recursive, hierarchical structure. It is probably significant that Algol 60 was also the first language to be defined, syntactically at least, with a context-free grammar. It is straightforward to define recursive structures using context-free grammars — specifically, a grammar written in Backus-Naur Form (BNF) for Algol 60. The basis of Algol 60 syntax can be written in extended BNF as follows:

$$\begin{array}{rcl}
\texttt{PROG} & \rightarrow & \texttt{BLOCK} \\
\texttt{BLOCK} & \rightarrow & \texttt{"begin"}\ \{\,\texttt{DECL}\,\}\ \{\,\texttt{STMT}\,\}\ \texttt{"end"} \\
\texttt{STMT} & \rightarrow & \texttt{BLOCK} \mid \cdots \\
\texttt{DECL} & \rightarrow & \texttt{FUNCDECL} \mid \cdots \\
\texttt{FUNCDECL} & \rightarrow & \texttt{HEAD}\ \texttt{BLOCK}
\end{array}$$

With blocks come ***nested scopes***. Earlier languages had ***local scope*** (for example, FORTRAN subroutines have local variables) but scopes were continuous (no “holes”) and not nested. It was not possible for one declaration to override another declaration of the same name. In Algol 60, however, an inner declaration overrides an outer declaration. The outer block in Listing 54 declares an integer `k`, which is first set to 6 and then incremented. The inner block declares another integer `k` which is set to 4 and then disappears without being used at the end of the block.

See Section 10.8.1 for further discussion of nesting.

## 11.2 Modules

By the early 70s, people realized that the “monolithic” structure of Algol-like programs was too restricted for large and complex programs. Furthermore, the “separate compilation” model of FORTRAN — and later C — was seen to be unsafe.

The new model that was introduced by languages such as Mesa and Modula divided a large program into modules.

**Listing 54:** Nested Declarations

```
begin
  integer k;
  k := 6;
  begin
    integer k;
    k := 1;
  end;
  k := k + 1;
end;
```

- ▷ A module is a meaningful unit (rather than a collection of possibly unrelated declarations) that may introduce constants, types, variables, and functions (called collectively ***features***).
- ▷ A module may ***import*** some or all of its features.
- ▷ In typical modular languages, modules can ***export*** types, functions, and variables.

**Example 41: Stack Module.** One module of a program might declare a stack module:

```
module Stack
{
  export initialize, push, pop, ....
  ....
}
```

Elsewhere in the program, we could use the stack module:

```
import Stack;
initialize();
push(67);
....
```

□

This form of module introduces a single instance: there is only one stack, and it is shared by all parts of the program that import it. In contrast, a stack template introduces a ***type*** from which we can create as many instances as we need.

**Example 42: Stack Template.** The main difference between the stack module and the stack template is that the template exports a type.

```
module Stack
{
  export type STACK, initialize, push, pop, ....
  ....
}
```

In other parts of the program, we can use the type `STACK` to create stacks.

```
import Stack;
STACK myStack;
initialize(myStack);
push(myStack, 67);
....
STACK yourStack;
....
```

□

The fact that we can declare many stacks is somewhat offset by the increased complexity of the notation: the name of the stack instance must appear in every function call. Some modular languages provide an alternative syntax in which the name of the instance appears before the function name, as if the module was a record. The last few lines of the example above would be written as:

```
import Stack;
STACK myStack;
```

```
    myStack.initialize();
    myStack.push(67);
    ....
```

### 11.2.1 Encapsulation

Parnas (1972) wrote a paper that made a number of important points.

▷ A large program will be implemented as a set of modules.

▷ Each programmer, or programming team, will be the ***owners*** of some modules and ***users*** of other modules.

▷ It is desirable that a programmer should be able to change the implementation of a module without affecting (or even informing) other programmers.

▷ This implies that a module should have an ***interface*** that is relatively stable and an ***implementation*** that changes as often as necessary. Only the owner of a module has access to its implementation; users do not have such access.

Parnas introduced these ideas as the "principle of information hiding". Today, it is more usual to refer to the goals as ***encapsulation***.

In order to use the module correctly without access to its implementation, users require a ***specification*** written in an appropriate language. The specification answer the question "What does this module do?" and the implementation answers the question "How does this module work?"

Most people today accept Parnas's principle, even if they do not apply it well. It is therefore important to appreciate how radical it was at the time of its introduction.

> [Parnas] has proposed a still more radical solution. His thesis is that the programmer is most effective if shielded from, rather than exposed to, the details of construction of system parts other than his own. This presupposes that all interfaces are completely and precisely defined. While that is definitely sound design, dependence upon its perfect accomplishment is a recipe for disaster. A good information system both exposes interface errors and stimulates their correction. (Brooks 1975, page 78)

Brooks wrote this in the first edition of *The Mythical Man Month* in 1975. Twenty years later, he had changed his mind.

> Parnas was right, and I was wrong. I am now convinced that information hiding, today often embodied in object-oriented programming, is the only was of raising the level of software design. (Brooks 1995, page 272)

Parnas's paper was published in 1972, only shortly after Wirth's (1971) paper on program development by stepwise refinement. Yet Parnas's approach is significantly different from Wirth's. Whereas Wirth assumes that a program can be constructed by filling in details, perhaps with occasional backtracking to correct an error, Parnas sees programming as an iterative process in which revisions and alterations are part of the normal course of events. This is a more "modern" view and certainly a view that reflects the actual construction of large and complex software systems.

## 11.3 Control Structures

Early PLs, such as FORTRAN, depended on three control structures:

- ▷ sequence;
- ▷ transfer (`goto`);
- ▷ call/return.

These structures are essentially those provided by assembly language. The first contribution of PLs was to provide names for variables and infix operators for building expressions.

The "modern" control structures first appeared in Algol 60. The most significant contribution of Algol 60 was the ***block***: a program unit that could contain data, functions, and statements. Algol 60 also contributed the familiar `if-then-else` statement and a rather complex loop statement. However, Algol 60 also provided `goto` statements and most programmers working in the sixties assumed that `goto` was essential.

The storm of controversy raised by Dijkstra's (1968) letter condemning the `goto` statement indicates the extent to which programmers were wedded to the `goto` statement. Even Knuth (1974) entered the fray with arguments supporting the `goto` statement in certain circumstances, although his paper is well-balanced overall. The basic ideas that Dijkstra proposed were as follows.

- ▷ There should be one flow into, and one flow out of, each control structure.
- ▷ All programming needs can be met by three structures satisfying the above property: the sequence; the loop with preceding test (`while-do`); and the conditional (`if-then-else`).

Dijkstra's proposals had previously been justified in a formal sense by Bohm and Jacopini (1966). This paper establishes that any program written with `goto` statements can be rewritten as an equivalent program that uses sequence, conditional, and loop structures only; it may be necessary to introduce Boolean variables. Although the result is interesting, Dijkstra's main point was not to transform old, `goto`-ridden programs but to write new programs using simple control structures only.

Despite the controversy, Dijkstra's arguments had the desired effect. In languages designed after 1968, the role of `goto` was down-played. Both C and Pascal have `goto` statements but they are rarely used in well-written programs. C, with `break`, `continue`, and `return`, has even less need for `goto` than Pascal. Ada has a `goto` statement because it was part of the US DoD requirements but, again, it is rarely needed.

The use of control structures rather than `goto` statements has several advantages.

- ▷ It is easier to read and maintain programs. Maintenance errors are less common.
- ▷ Precise reasoning (for example, the use of formal semantics) is simplified when `goto` statements are absent.
- ▷ Certain optimizations become feasible because the compiler can obtain more precise information in the absence of `goto` statements.

### 11.3.1 Loop Structures.

The control structure that has elicited the most discussion is the loop. The basic loop has a single test that precedes the body:

**Listing 55:** Reading in Pascal

```
var n, sum :  int;
sum := 0;
read(n);
while n ≥ 0 do
  begin
    sum := sum + n;
    read(n);
  end
```

**Listing 56:** Reading in C

```
int sum = 0;
while (true)
{
  int n;
  cin >> n;
  if (n < 0)
    break;
  sum += n;
}
```

`while` $E$ `do` $S$

It is occasionally useful to perform the test after the body: Pascal provides `repeat/until` and C provides `do/while`.

This is not quite enough, however, because there are situations in which it is useful to exit from a loop from within the body. Consider, for example, the problem of reading and adding numbers, terminating when a negative number is read without including the negative number in the sum. Listing 55 shows the code in Pascal. In C and C++, we can use `break` to avoid the repeated code, as shown in Listing 56.

Some recent languages, recognizing this problem, provide only one form of loop statement consisting of an unconditional repeat and a conditional exit that can be used anywhere in the body of the loop.

### 11.3.2 Procedures and Functions

Some languages, such as Ada and Pascal, distinguish the terms `procedure` and `function`. Both names refer to "subroutines" — program components that perform a specific task when invoked from another component of the program. A ***procedure*** is a subroutine that has some effect on the program data but does not return a result. A ***function*** is a subroutine that returns a result; a function may have some effect on program data, called a "side effect", but this is often considered undesirable.

Languages that make a distinction between procedures and functions usually also make a distinction between "actions" and "data". For example, at the beginning of the first Pascal text (Jensen and Wirth 1976), we find the following statement:

> An ***algorithm*** or computer program consists of two essential parts, a description of ***actions*** which are to be performed, and a description of the ***data***, which are manipulated by the actions. Actions are described be so-called ***statements***, and data are described by so-called ***declarations*** and ***definitions***.

In other languages, such as Algol and C, the distinction between "actions" and "data" is less emphasized. In these languages, every "statement" has a value as well as an effect. Consistently, the distinction between "procedure" and "function" also receives less emphasis.

C, for example, does not have keywords corresponding to `procedure` and `function`. There is a single syntax for declaring and defining all functions and a convention that, if a function does not return a result, it is given the return type `void`.

We can view the progression here as a separation of concerns. The "concerns" are: whether a subroutine returns a value; and whether it has side-effects. Ada and Pascal combine the concerns but, for practical reasons, allow "functions with side-effects". In C, there are two concerns: "having a value" and "having an effect" and they are essentially independent.

**Example 43: Returning multiple results.** Blue (Kölling 1999) has a uniform notation for returning either one or more results. For example, we can define a routine with this heading:

```
findElem (index: Integer) -> (found: Boolean, elem: Element) is ....
```

The parameter `index` is passed to the routine `findElem`, and the routine returns two values: `found` to indicate whether something was found, and `elem` as the object found. (If the object is not found, `elem` is set to the special value `nil`.)

Adding multiple results introduces the problem of how to use the results. Blue also provides ***multi-assignments***: we can write

```
success, thing := findElem("myKey")
```

Multi-assignments, provided they are implemented properly, provide other advantages. For example we can write the standard "swap sequence"

```
t := a;
a := b;
b := t;
```

in the more concise and readable form

```
a, b := b, a
```

□

**Exercise 39.** Describe a "proper" implementation of the multi-assignment statement. Can you think of any other applications of multi-assignment? □

**Exercise 40.** Using Example 43 and other examples, discuss the introduction of new features into PLs. □

### 11.3.3 Exceptions.

The basic control structures are fine for most applications provided that nothing goes wrong. Although in principle we can do everything with conditions and loops, there are situations in which these constructs are inadequate.

**Example 44: Matrix Inversion.** Suppose that we are writing a function that inverts a matrix. In almost all cases, the given matrix is non-singular and the inversion proceeds normally. Occasionally, however, a singular matrix is passed to the function and, at some point in the calculation, a division by zero will occur. What choices do we have?

- ▷ We could test the matrix for singularity before starting to invert it. Unfortunately, the test involves computing the determinant of the matrix, which is almost as much work as inverting it. This is wasteful, because we know that most of the matrices that we will receive are non-singular.
- ▷ We could test every divisor before performing a division. This is wasteful if the matrix is unlikely to be singular. Moreover, the divisions probably occur inside nested loops: each level of loop will require Boolean variables and exit conditions that are triggered by a zero divisor, adding overhead.
- ▷ Rely on the hardware to signal an ***exception*** when division by zero occurs. The exception can be caught by a ***handler*** that can be installed at any convenient point in the calling environment.

□

PL/I (Section 4.5) was the first major language to provide exception handling. C provided primitives (`setjmp`/`longjmp`) that made it possible to implement exceptin handling. For further descriptions of exception handling in particular languages, see Section 4.

# 12 Issues in OOP

OOP is the programming paradigm that currently dominates industrial software development. In this section, we discuss various issues that arise in OOP.

## 12.1 Algorithms + Data Structures = Programs

The distinction between code and data was made at an early stage. It is visible in FORTRAN and reinforced in Algol 60. Algol provided the recursive block structure for code but almost no facilities for data structuring. Algol 68 and Pascal provided recursive data structures but maintained the separation between code and data.

LISP brought algorithms (in the form of functions) and data structures together, but:

- ▷ the mutability of data was downplayed;
- ▷ subsequent FPLs reverted to the Algol model, separating code and data.

Simula started the move back towards the von Neumann Architecture at a higher level of abstraction than machine code: the computer was recursively divided into smaller computers, each with its own code and stack. This was the feature of Simula that attracted Alan Kay and led to Smalltalk (Kay 1996).

Many OOPLs since Smalltalk have withdrawn from its radical ideas. C++ was originally called "C with classes" (Stroustrup 1994) and modern C++ remains an Algol-like systems programming language that supports OOP. Ada has shown even more reluctance to move in the direction of OOP, although Ada 9X has a number of OO features.

## 12.2 Values and Objects

Some data behave like mathematical objects and other data behave like representations of physical objects (MacLennan 1983).

Consider the sets $\{1, 2, 3\}$ and $\{2, 3, 1\}$. Although these sets have different representations (the ordering of the members is different), they have the same value because in set theory the only important property of a set is the members that it contains: "two sets are equal if and only if they contain the same members".

Consider two objects representing people:

```
[name="Peter", age=55]
[name="Peter", age=55]
```

Although these objects have the same value, they might refer to different people who have the same name and age. The consequences of assuming that these two objects denote the same person could be serious: in fact, innocent people have been arrested and jailed because they had the misfortune to have the same name and social security number as a criminal.

In the following comparison of values and objects, statements about values on the left side of the page are balanced by statements about objects on the right side of the page.

**Values**

**Objects**

Integers have values. For example, the integer 6 is a value.

Objects possess values as attributes. For example, a counter might have 6 as its current value.

Values are abstract. The value 7 is the common property shared by all sets with cardinality 7.

Objects are concrete. An object is a particular collection of subobjects and values. A counter has one attribute: the value of the count.

Values are immutable (i.e., do not change).

Objects are usually, but not necessarily, mutable (i.e., may change).

It does not make sense to change a value. If we change the value 3, it is no longer 3.

It is meaningful, and often useful, to change the value of an object. For example, we can increment a counter.

Values need representations. We need representations in order to compute with values. The representations 7, VII, *sept*, *sju*, and a particular combination of bits in the memory of a computer are different representations of the same abstract value.

Objects need representations. An object has subobjects and values as attributes.

It does not make sense to talk about the identity of a value. Suppose we evaluate $2 + 3$, obtaining 5, and then $7 - 2$, again obtaining 5. Is it the same 5 that we obtain from the second calculation?

Identity is an important attribute of objects. Even objects with the same attributes may have their own, unique identities: consider identical twins.

We can copy a representation of a value, but copying a value is meaningless.

Copying an object certainly makes sense. Sometimes, it may even be useful, as when we copy objects from memory to disk, for example. However, multiple copies of an object with a supposedly unique identity can cause problems, known as "integrity violations" in the database community.

We consider different representations of the same value to be equal.

An object is equal to itself. Whether two distinct objects with equal attributes are equal depends on the application. A person who is arrested because his name and social insurance number is the same as those of a wanted criminal may not feel "equal to" the criminal.

Aliasing is not a problem for values. It is immaterial to the programmer, and should not even be detectable, whether two instances of a value are stored in the same place or in different places.

All references to a particular object should refer to the same object. This is sometimes called "aliasing", and is viewed as harmful, but it is in fact desirable for objects with identity.

Functional and logical PLs usually provide values.

Object oriented PLs should provide objects.

Programs that use values only have the property of referential transparency.

Programs that provide objects do not provide referential transparency, but may have other desirable properties such as encapsulation.

Small values can be held in machine words. Large values are best represented by references (pointers to the data structure representing the value.

Very small objects may be held in machine words. Most objects should be represented by references (pointers to the data structure holding the object.

The assignment operator for values should be a reference assignment. There is no need to copy values, because they do not change.

The assignment operator for objects should be a reference assignment. Objects should not normally be copied, because copying endangers their integrity.

The distinction between values and objects is not as clear as these comparisons suggest. When we look at programming problems in more deatil, doubts begin to arise.

**Exercise 41.** Should sets be values? What about sets with mutable components? Does the model handle objects with many to one abstraction functions? What should a PL do about values and objects? □

Conclusions:

- ▷ The CM must provide values, may provide objects.
- ▷ What is "small" and what is "large"? This is an implementation issue: the compiler must decide.
- ▷ What is "primitive" and what is "user defined"? The PL should minimize the distinction as much as possible.
- ▷ Strings should behave like values. Since they have unbounded size, many PLs treat them as objects.
- ▷ Copying and comparing are ***semantic*** issues. Compilers can provide default operations and "building bricks", but they cannot provide fully appropriate implementations.

## 12.3 Classes *versus* Prototypes

The most widely-used OOPLs — C++, Smalltalk, and Eiffel — are class-based. Classes provide a visible program structure that seems to be reassuring to designers of complex software systems.

Prototype languages are a distinct but active minority. The relation between classes and prototypes is somewhat analogous to the relation between procedural (FORTRAN and Algol) and functional (LISP) languages in the 60s.

Prototypes provide more flexibility and are better suited to interactive environments than classes. Although large systems have been built using prototype languages, these languages seem to be more often used for exploratory programming.

## 12.4 Types *versus* Objects

In most class-based PLs, functions are associated with objects. This has the effect of making binary operations asymmetric. For example, $x = y$ is interpreted as `x.equal(y)` or, approximately, "send the message `equal` with argument `y` to the object `x`". It is not clear whether, with this interpretation, $y = x$ means the same as $x = y$. The problem is complicated further by inheritance: if class $C$ is a subclass of class $P$, can we compare an instance of $C$ to an instance of $P$, and is this comparison commutative?

Some of the problems can be avoided by associating functions with types rather than with objects. This is the approach taken by CLU and some of the "functional" OOPLs. An expression such as $x = y$ is interpreted in the usual mathematical sense as `equal(x,y)`. However, even this approach has problems if $x$ and $y$ do not have the same type.

## 12.5 Pure *versus* Hybrid Languages

Smalltalk, Eiffel, and Java are ***pure*** OOPLs: they can describe only systems of objects and programmers are forced to express everything in terms of objects.

C++ is a ***hybrid*** PL in that it supports both procedural and object styles. Once the decision was made to extend C rather than starting afresh (Stroustrup 1994, page 43), it was inevitable that C++ could be a hybrid language. In fact, its precursor, "C with classes", did not even claim to be object oriented.

A hybrid language can be dangerous for inexperienced programmers because it encourages a hybrid programming style. OO design is performed perfunctorily and design problems are solved by reverting to the more general, but lower level, procedural style. (This is analogous to the early days of "high level" programming when programmers would revert to assembler language to express constructs that were not easy or possible to express at a high level.)

For experienced programmers, and especially those trained in OO techniques, a hybrid language is perhaps better than a pure OOPL. These programmers will use OO techniques most of the time and will only resort to procedural techniques when these are clearly superior to the best OO solution. As OOPLs evolve, these occasions should become rarer.

Perhaps "pure or hybrid" is the wrong question. A better question might be: suppose a PL provides OOP capabilities. Does it need to provide anything else? If so, what? One answer is: FP capabilities. The multi-paradigm language LEDA (Budd 1995) demonstrates that it is possible to provide functional, logic, and object programming in a single language. However, LEDA provides these in a rather *ad hoc* way. It more tightly-integrated design would be preferable, but is harder to accomplish.

## 12.6 Closures *versus* Classes

The most powerful programming paradigms developed so far are those that do not separate code and data but instead provide "code and data" units of arbitrary size. This can be accomplished by high order functions and mutable state, as in Scheme (Section 5.2) or by classes. Although each method has advantages for certain applications, the class approach seems to be better in most practical situations.

## 12.7 Inheritance

Inheritance is one of the key features of OOP and at the same time one of the most troublesome. Conceptually, inheritance can be defined in a number of ways and has a number of different uses. In practice, most (but not all!) OOPLs provide a single mechanism for inheritance and leave it to the programmer to decide how to use the mechanism. Moreover, at the ***implementation*** level, the varieties of inheritance all look much the same — a fact that discourages PL designers from providing multiple mechanisms.

One of the clearest examples of the division is Meyer's (1988) "marriage of convenience" in which the class `ArrayStack` inherits from `Array` and `Stack`. The class `ArrayStack` behaves like a stack — it has functions such as `push`, `pop`, and so on — and is represented as an array. There are other ways of obtaining this effect — notably, the class `ArrayStack` could have an array object as an attribute — but Meyer maintains that inheritance is the most appropriate technique, at least in Eiffel.

Whether or not we agree with Meyer, it is clear that there two different relations are involved. The relation `ArrayStack` $\sim$ `Stack` is not the same as the relation `ArrayStack` $\sim$ `Array`. Hence the question: should an OOPL distinguish between these relations or not?

Based on the "marriage of convenience" the answer would appear to be "yes" — two distinct ends are achieved and the means by which they are achieved should be separated. A closer examination, however, shows that the answer is by no means so clear-cut. There are many practical programming situations in which code in a parent class can be reused in a child class even when interface consistency is the primary purpose of inheritance.

## 12.8 A Critique of C++

C++ is both the most widely-used OOPL and the most heavily criticized. Further criticism might appear to be unnecessary. The purpose of this section is to show how C++ fails to fulfill its role as the preferred OOPL.

The first part of the following discussion is based on a paper by LaLonde and Pugh (1995) in which they discuss C++ from the point of view of a programmer familiar with the principles of OOP, exemplified by a language such as Smalltalk, but who is not familiar with the details of C++.

We begin by writing a specification for a simple `BankAccount` class with an owner name, a balance, and functions for making deposits and withdrawals: see Listing 57.

▷ Data members are prefixed with "`p`" to distinguish them from parameters with similar names.

▷ The instance variable `pTrans` is a pointer to an array containing the transactions that have taken place for this account. Functions that operate on this array have been omitted.

We can create instances of `BankAccount` on either the stack or the heap:

```
BankAccount account1("John"); // on the stack
BankAccount *account2 = new BankAccount("Jane"); // on the heap
```

The first account, `account1`, will be deleted automatically at the end of the scope containing the declaration. The second account, `account2`, will be deleted only when the program executes a `delete` operation. In both cases, the destructor will be invoked.

The notation for accessing functions depends on whether the variable refers to the object directly or is a pointer to the object.

```
account1.balance()
account2→balance()
```

A member function can access the private parts of another instance of the same class, as shown in Listing 58. This comes as a surprise to Smalltalk programmers because, in Smalltalk, an object can access only its own components.

The next step is to define the member functions of the class `BankAccount`: see Listing 59.

Listing 60 shows a simple test of the class `BankAccount`. This test introduces two bugs.

- The array `account1.pTrans` is lost because it is over-written by `account2.pTrans`.
- At the end of the test, `account2.pTrans` will be deleted twice, once when the destructor for `account1` is called, and again when the destructor for `account2` is called.

Although most introductions to C++ focus on stack objects, it is more interesting to consider heap objects: Listing 61.

The code in Listing 61 — creating three special accounts — is rather specialized. Listing 62 generalizes it so that we can enter the account owner's names interactively.

The problem with this code is that all of the bank accounts have the same owner: that of the last name entered. This is because each account contains a pointer to the unique `buffer`. We can fix this by allocating a new buffer for each account, as in Listing 63.

This version works, but is poorly designed. Deleting the owner string should be the responsibility of the object, not the client. The destructor for `BankAccount` should delete the owner string. After making this change, we can remove the destructor for `owner` in the code above: Listing 64.

Unfortunately, this change introduces another problem. Suppose we use the loop to create two accounts and then create the third account with this statement:

```
accounts[2] = new BankAccount("Tina");
```

**Listing 57:** Declaring a Bank Account

```
class BankAccount
{
public:
  BankAccount(char * newOwner = "");
  ~BankAcocunt();
  void owner(char *newOwner);
  char *owner();
  double balance();
  void credit(double amount);
  void debit(double amount);
  void transfer(BankAccount *other, double amount);
private:
  char *pOwner;
  double pBalance;
  Transaction *pTrans[MAXTRANS];
};
```

**Listing 58:** Accessing Private Parts

```
void BankAccount::transfer(BankAccount *other, double amount)
{
  pBalance += amount;
  other→pBalance -= amount;
}
```

**Listing 59:** Member Functions for the Bank Account

```
BankAccount::BankAccount(char *newOwner)
{
  pOwner = newOwner;
  pBalance = 0.0;
}
BankAccount::∼BankAccount()
{
  // Delete individual transactions
  delete [] pTrans; // Delete the array of transactions
}
void BankAccount owner(char *newOwner)
{
  pOwner = newOwner;
}
char *BankAccount owner()
{
  return pOwner;
}
....
```

**Listing 60:** Testing the Bank Account

```
void test
{
  BankAccount account1("Anne");
  BankAccount account2("Bill");
  account1 = acount2;
}
```

**Listing 61:** Using the Heap

```
BankAccount *accounts[] = new BankAccount * [3];
accounts[0] = new BankAccount("Anne");
accounts[1] = new BankAccount("Bill");
accounts[2] = new BankAccount("Chris");
// Account operations omitted.
// Delete everything.
for (int i = 0; i < 3; i++)
  delete accounts[i];
delete accounts;
```

**Listing 62:** Entering a Name

```
BankAccount *accounts[] = new BankAccount * [3];
char * buffer = new char[100];
for (int i = 0; i < 3; i++)
{
  cout << "Please enter your name:  ";
  cin.getline(buffer, 100);
  accounts[i] = new BankAccount(buffer);
}
// Account operations omitted.
// Delete everything.
for (int i = 0; i < 3; i++)
  delete accounts[i];
delete accounts;
delete buffer;
```

**Listing 63:** Adding buffers for names

```
// char * buffer = new char[100]; (deleted)
for (int i = 0; i < 3; i++)
{
  cout << "Please enter your name:  ";
  char * buffer = new char[100];
  cin.getline(buffer, 100);
  accounts[i] = new BankAccount(buffer);
}
// Account operations omitted.
// Delete everything.
for (int i = 0; i < 3; i++)
{
  delete accounts[i]→owner();
  delete accounts[i];
}
delete accounts;
delete buffer;
```

**Listing 64:** Correcting the Destructor

```
BankAccount::∼BankAccount()
{
  delete pOwner;
  // Delete individual transactions
  delete [] pTrans; // Delete the array of transactions
}
```

**Listing 65:** Correcting the Constructor

```
BankAccount::BankAccount(char *newOwner)
{
char *pOwner = new char (strlen(newOwner) + 1);
strcpy(pOwner, newOwner);
}
```

**Listing 66:** Correcting the Owner Function

```
void BankAccount owner(char *newOwner)
{
  delete pOwner;
  pOwner = new char(strlen(newOwner) + 1);
  strcpy(pOwner, newOwner);
}
```

The account destructor fails because `"Tina"` was not allocated using `new`. To correct this problem, we must modify the constructor for `BankAccount`. The rule we must follow is that data must be allocated with the constructor and deallocated with the destructor, or controlled entirely outside the object. Listing 65 shows the new constructor.

The next problem occurs when we try to use the overloaded function `owner()` to change the name of an account owner.

```
accounts[2]→owner("Fred");
```

Once again, we have introduced two bugs:

- The original owner of `accounts[2]` will be lost.
- When the destructor is invoked for `accounts[2]`, it will attempt to delete `"Fred"`, which was not allocated by `new`. To correct this problem, we must rewrite the member function `owner()`, as shown in Listing 66.

We now have bank accounts that work quite well, provided that they are allocated on the heap. If we try to mix heap objects and stack objects, however, we again encounter difficulties: Listing 67. The error occurs because `transfer` expects the address of an account rather than an actual account. We correct it by introducing the reference operator, `&`.

```
account1→transfer(&account3, 10.0);
```

We could make things easier for the programmer who is using accounts by declaring an overloaded version of `transfer()` as shown below. Note that, in the version of Listing 68, we must use the

**Listing 67:** Stack and Heap Allocation

```
BankAccount *account1 = new BankAccount("Anne");
BankAccount *account2 = new BankAccount("Bill");
BankAccount account3("Chris");
account1→transfer(account2, 10.0); // OK
account1→transfer(account3, 10.0); // Error!
```

**Listing 68:** Revising the Transfer Function

```
        void BankAccount::transfer(BankAccount &other, double amount)
        {
          pBalance += amount;
          other.pBalance -= amount;
        }
```

**Listing 69:** Copying

```
        BankAccount test(BankAccount account)
        {
        return account;
        }
        BankAccount myAccount;
        myAccount = test(account2);
```

operator “`.`” rather than “→”. This is because `other` behaves like an object, although in fact an address has been passed.

The underlying problem here is that references (`&`) work best with stack objects and pointers (`*`) work best with heap objects. It is awkward to mix stack and heap allocation in C++ because two sets of functions are needed.

If stack objects are used without references, the compiler is forced to make copies. To implement the code shown in Listing 69, C++ uses the (default) copy constructor for `BankAccount` twice: first to make a copy of `account2` to pass to `test()` and, second, to make another copy of `account2` that will become `myAccount`. At the end of the scope, these two copies must be deleted.

The destructors will fail because the default copy constructor makes a shallow copy — it does not copy the `pTrans` fields of the accounts. The solution is to write our own copy constructor and assignment overload for class `BankAccount`, as in Listing 70. The function `pDuplicate()` makes a deep copy of an account the function; it must make copies of all of the components of a back account object and destroy all components of the old object. The function `pDelete` has the same effect as the destructor. It is recommended practice in C++ to write these functions for any class that contains pointers.

LaLonde and Pugh continue with a discussion of infix operators for user-defined classes. If the objects are small — complex numbers, for example — it is reasonable to store the objects on the stack and to use call by value. C++ is well-suited to this approach: storage management is automatic and the only overhead is the copying of values during function call and return.

If the objects are large, however, there are problems. Suppose that we want to provide the standard algebraic operations $(+, -, *)$ for $4 \times 4$ matrices. (This would be useful for graphics programming, for example.) Since a matrix occupies $4 \times 4 \times 8 = 128$ bytes, copying a matrix is relatively expensive and the stack approach will be inefficient. But if we write functions that work with pointers to matrices, it is difficult to manage allocation and deallocation. Even a simple function call such as

```
        project(A + B);
```

will create a heap object corresponding to `A + B` but provides no opportunity to delete this object. (The function cannot delete it, because it cannot tell whether its argument is a temporary.) The same problem arises with expressions such as

**Listing 70:** Copy Constructors

```
BankAccount::BankAccount(BankAccount &account)
{
  pDuplicate(account);
}
BankAccount & BankAccount operator = (BankAccount &rhs)
{
  if (this == &rhs)
    return this;
  pDelete();
  pDuplicate(rhs);
  return *this;
}
```

**Listing 71:** Using extended assignment operators

```
A = C;
A *= D;
A += B;
```

```
A = B + C * D;
```

It is possible to implement the extended assignment operators, such as `+=`, without losing objects. But this approach reduces us to a kind of assembly language style of programming, as shown in Listing 71.

These observations by LaLonde and Pugh seem rather hostile towards C++. They are making a comparison between C++ and Smalltalk based only on ease of programming. The comparison is one-sided because they do not discuss factors that might make C++ a better choice than Smalltalk in some circumstances, such as when efficiency is important. Nevertheless, the discussion makes an important point: the "stack model" (with value semantics) is inferior to the "heap model" (with reference semantics) for many OOP applications. It is significant that most language designers have chosen the heap model for OOP (Simula, Smalltalk, CLU, Eiffel, Sather, Dee, Java, Blue, amongst others) and that C++ is in the minority on this issue.

Another basis for criticizing C++ is that is has become, perhaps not intentionally, a hacker's language. A sign of good language design is that the features of the language are used most of the time to solve the problems that they were intended to solve. A sign of a hacker's language is that language "gurus" introduce "hacks" that solve problems by using features in bizarre ways for which they were not originally intended.

Input and output in C++ is based on "streams". A typical output statement looks something like this:

```
cout << "The answer is " << answer << '.'  << endl;
```

The operator `<<` requires a stream reference as its left operand and an object as its right operand. Since it returns a stream reference (usually its left operand), it can be used as an infix operator in compound expressions such as the one shown. The notation seems clear and concise and, at first glance, appears to be an elegant way of introducing output and input (with the operator `>>`) capabilities.

**Listing 72:** A Manipulator Class

```
class ManipulatorForInts
{
  friend ostream & operator << (ostream & const ManipulatorForInts &);
public:
  ManipulatorForInts (int (ios::*)(int), int);
  ....
private:
  int (ios::*memberfunc)(int);
  int value;
}
```

Obviously, `<<` is heavily overloaded. There must be a separate function for each class for which instances can be used as the right operand. (`<<` is also overloaded with respect to its left operand, because there are different kinds of output streams.) In the example above, the first three right operands are a string, a `double`, and a character.

The first problem with streams is that, although they are superficially object oriented (a stream is an object), they do not work well with inheritance. Overloading provides ***static*** polymorphism which is incompatible with ***dynamic*** polymorphism (recall Section 10.6). It is possible to write code that uses dynamic binding but only by providing auxiliary functions.

The final operand, `endl`, writes end-of-line to the stream and flushes it. It is a global function with signature

```
ios & endl (ios & s)
```

A programmer who needs additional "manipulators" of this kind has no alternative but to add them to the global name space.

Things get more complicated when formatting is added to streams. For example, we could change the example above to:

```
cout << "The answer is " << setprecision(8) << answer << '.'  << endl;
```

The first problem with this is that it is verbose. (The verbosity is not really evident in such a small example, but it is very obvious when streams are used, for example, to format a table of heterogeneous data.) The second, and more serious problem, is the implementation of `setprecision`. The way this is done is (roughly) as follows (Teale 1993, page 171). First, a class for manipulators with integer arguments is declared, as in Listing 72, and the inserter function is defined:

```
ostream & operator << (ostream & os, const ManipulatorForInts & m)
{
  (os.*memberfunc)(m.value);
  return os;
}
```

Second, the function that we need can be defined:

```
ManipulatorForInts setprecision (int n)
{
  return ManipulatorForInts (& ios::precision, n);
}
```

**Listing 73:** File handles as Booleans

```
        ifstream infile ("stats.dat");
        if (infile) ....
        if (!infile) ....
```

It is apparent that a programmer must be quite proficient in C++ to understand these definitions, let alone design similar functions. To make matters worse, the actual definitions are not as shown here, because that would require class declarations for many different types, but are implemented using templates.

There are many other tricks in the implementation of streams. For example, we can use file handles as if they had boolean values, as in Listing 73. It seems that there is a boolean value, `infile`, that we can negate, obtaining `!infile`. In fact, `infile` can be used as a conditional because there is a function that converts `infile` to `(void *)`, with the value `NULL` indicating that the file is not open, and `!infile` can be used as a conditional because the operator `!` is overloaded for class `ifstream` to return an `int` which is 0 if the file is open.

These are symptoms of an even deeper malaise in C++ than the problems pointed out by authors of otherwise excellent critiques such as (Joyner 1992; Sakkinen 1992). Most experienced programmers can get used to the well-known idiosyncrasies of C++, but few realize that frequently-used and apparently harmless constructs are implemented by subtle and devious tricks. Programs that look simple may actually conceal subtle traps.

## 13 Conclusion

**Values and Objects.** Mathematically oriented views of programming favour values. Simulation oriented views favour objects. Both views are useful for particular applications. A PL that provides both in a consistent and concise way might be simple, expressive, and powerful.

The "logical variable" (a variable that is first unbound, then bound, and later may be unbound during backtracking) is a third kind of entity, after values and objects. It might be fruitful to design a PL with objects, logical variables, unification, and backtracking. Such a language would combine the advantages of OOP and logic programming.

**Making Progress.** It is not hard to design a large and complex PL by throwing in numerous features. The hard part of design is to find simple, powerful abstractions — to achieve more with less. In this respect, PL design resembles mathematics. The significant advances in mathematics are often *simplifications* that occur when structures that once seemed distinct are united in a common abstraction. Similar simplifications have occurred in the evolution of PLs: for example, Simula (Section 6.1) and Scheme (Section 5.2). But programs are ***not*** mathematical objects. A rigorously mathematical approach can lead all too rapidly to the "Turing tar pit".

**Practice and Theory.** The evolution of PLs shows that, most of the time, practice leads theory. Designers and implementors introduce new ideas, then theoreticians attempt to what they did and how they could have done it better. There are a number of PLs that have been based on purely theoretical principles but few, if any, are in widespread use.

Theory-based PLs are important because they provide useful insights. Ideally, they serve as testing environments for ideas that may eventually be incorporated into new mainstream PLs. Unfortunately, it is often the case that they show retroactively how something should have been done when it is already too late to improve it.

**Multiparadigm Programming.** PLs have evolved into several strands: procedural programming, now more or less subsumed by object oriented programming; functional programming; logic programming and its successor, constraint programming. Each paradigm performs well on a particular class of applications. Since people are always searching for "universal" solutions, it is inevitable that some will try to build a "universal" PL by combining paradigms. Such attempts will be successful to the extent that they achieve overall simplification. It is not clear that any future universal language will be accepted by the programming community; it seems more likely that specialized languages will dominate in the future.

# A Abbreviations and Glossary

**Actual parameter.** A value passed to a function.

**ADT.** Abstract data type. A data type described in terms of the operations that can be performed on the data.

**AR.** Activation record. The run-time data structure corresponding to an Algol 60 block. Section 4.3.

**Argument.** A value passed to a function; synonym of ***actual parameter***.

**Bind.** Associate a property with a name.

**Block.** In Algol 60, a syntactic unit containing declarations of variables and functions, followed by code. Successors of Algol 60 define blocks in similar ways. In Smalltalk, a block is a ***closure*** consisting of code, an environment with bindings for local variables, and possibly parameters.

**BNF.** Backus-Naur Form (originally Backus Normal Form). A notation for describing a context-free grammar, introduced by John Backus for describing Algol 60. Section 3.4.2.

**Call by need.** The argument of a function is not evaluated until its value is needed, and then only once. See ***lazy evaluation***.

**CBN.** Call by name. A parameter passing method in which actual parameters are evaluated when they are needed by the called function. Section 4.3

**CBV.** Call by value. A parameter passing method in which the actual parameter is evaluated before the function is called.

**Closure.** A data structure consisting of one (or more, but usually one) function and its lexical environment. Section 5.2.

**CM.** Computational Model. Section 9.2

**Compile.** Translate source code (program text) into code for a physical (or possibly abstract) machine.

**CS.** Computer Science.

**Delegation.** An object that accepts a request and then forwards the request to another object is said to "delegate" the request.

**Dummy parameter.** ***Formal parameter*** (FORTRAN usage).

**Dynamic.** Used to describe something that happens during execution, as opposed to ***static***.

**Dynamic Binding.** A binding that occurs during execution.

**Dynamic Scoping.** The value of a non-local variable is the most recent value that has been assigned to it during the execution of the program. Cf. Lexical Scoping. Section 5.1.

**Dynamic typing.** Performing type checking "on the fly", while the program is running.

**Eager evaluation.** The arguments of a function are evaluated at the time that it is called, whether or not their values are needed.

**Early binding.** Binding a property to a name at an early stage, usually during compilation.

**Exception handling.** A PL feature that permits a computation to be abandoned; control is transferred to a handler in a possibly non-local scope.

**Extent.** The time during execution during which a variable is active. It is a dynamic property, as opposed to ***scope***.

**Formal parameter.** A variable name that appears in a function heading; the variable will receive a value when the function is called. "Parameter" without qualification usually means "formal parameter".

**FP(L).** Functional Programming (Language). Section 5.

**Garbage collection.** Recycling of unused memory when performed automatically by the implementatin rather than coded by the programmer. (The term "garbage collection" is widely used but inaccurate. The "garbage" is thrown away: it is the useful data that is collected!)

**GC.** ***Garbage collection***.

**Global scope.** A name that is accessible throughout the (static) program text has global scope.

**Heap.** A region for storing variables in which no particular order of allocation or deallocation is defined.

**High order function.** A function that either accepts a function as an argument, or returns a function as a value, or both.

**Higher order function.** Same as ***high order function***.

**Inheritance.** In OOPLs, when an object (or class) obtains some of its features from another object (or class), it is said to ***inherit*** those features.

**Interpret.** Execute a program one statement at a time, without translating it into machine code.

**Lambda calculus.** A notation for describing functions, using the Greek letter $\lambda$, introduced by Church.

**Late binding.** A binding (that is, attachment of a property to a name) that is performed at a later time than would normally be expected. For example, procedural languages bind functions to names during compilation, but OOPLs "late bind" function names during execution.

**Lazy evaluation.** If the implementation uses lazy evaluation, or ***call by need***, a function does not evaluate an argument until the value of the argument is actually required for the computation. When the argument has been evaluated, its value is stored and it is not evaluated again during the current invocation of the function.

**Lexical Scoping.** The value of a non-local variables is determined in the static context as determined by the text of the program. Cf. Dynamic Scoping. Section 5.1.

**Local scope.** A name that is accessible in only a part of the program text has local scope.

**OOP(L).** Object Oriented Programming (Language). Section 6.

**Orthogonal.** Two language features are ***orthogonal*** if they can be combined without losing any of their important properties. Section 4.6.

**Overloading.** Using the same name to denote more than one entity (the entities are usually functions).

**Own variable.** A local variable in an Algol 60 program that has local scope (it can be accessed only within the procedure in which it is declared) but global extent (it is created when the procedure is first called and persists until the program termiantes). Section 4.3.

**Parameter.** Literally "unknown" or "unmeasurable" (from Greek). Used in PLs for the objects passed to functions.

**Partial function.** A function that is defined over only a part of its domain. $\sqrt{x}$ is partial over the domain of reals because it has no real value if $x < 0$.

**PL.** Programming Language.

**Polymorphism.** Using one name to denote several distinct objects. (Literally "many shapes").

**Recursion.** A process or object that is defined partly in terms of itself without circularity.

**Reference semantics.** Variable names denote objects. Used by all functional and logic languages, and most OOPLs.

**RE.** Regular Expression. Section 3.4.

**Scope.** The region of text in which a variable is accessible. It is a static property, as opposed to ***extent***.

**Semantic Model.** An alternative term for Computational Model (CM). Section 9.2.

**Semantics.** A system that assigns meaning to programs.

**Stack.** A region for storing variables in which the most recently allocated unit is the first one to be deallocated ("last-in, first out", LIFO). The name suggests a stack of plates.

**Static.** Used to describe something that happens during compilation (or possible during linking but, anyway, before the program is executed), as opposed to ***dynamic***.

**Static Binding.** A value that is bound before the program is executed. Cf. Dynamic Binding.

**Static typing.** Type checking performed during compilation.

**Strong typing.** Detect and report all type errors, as opposed to ***weak typing***.

**Template.** A syntactic construct from which many distinct objects can be generated. A template is different from a ***declaration*** that constructs only a single object. (The word "template" has a special sense in C++, but the usage is more or less compatible with our definition.)

**Thunk.** A function with no parameters used to implement the ***call by name*** mechanism of Algol 60 and a few other languages.

**Total function.** A function that is defined for all values of its arguments in its domain. $e^x$ is total over the domain of reals.

**Value semantics.** Variable names denote locations. Most imperative languages use value semantics.

**Weak typing.** Detect and report some type errors, but allow some possible type errors to remain in the program.

# References

Abelson, H. and G. Sussman (1985). *Structure and Interpretation of Computer Programs.* MIT Press.

Åke Wikström (1987). *Functional Programming Using Standard ML.* Prentice Hall.

Apt, K. R., J. Brunekreef, V. Partington, and A. Schaerf (1998, September). Alma-0: an imperative language that supports declarative programming. *ACM Trans. Programming Languages and Systems 20*(5), 1014–1066.

Arnold, K. and J. Gosling (1998). *The Java Programming Language* (Second ed.). The Java Series. Addison Wesley.

Ashcroft, E. and W. Wadge (1982). $R_x$ for semantics. *ACM Trans. Programming Languages and Systems 4*(2), 283–294.

Backus, J. (1957, February). The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference.*

Backus, J. (1978, August). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*(8), 613–64.

Bergin, Jr., T. J. and R. G. Gibson, Jr. (Eds.) (1996). *History of Programming Languages–II.* ACM Press (Addison Wesley).

Boehm, H.-J. and M. Weiser (1988). Garbage collection in an uncooperative environment. *Software: Practice & Experience 18*(9), 807–820.

Bohm, C. and G. Jacopini (1966, May). Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the ACM 9*(5), 366–371.

Bratko, I. (1990). *PROLOG: Programming for Artificial Intelligence* (Second ed.). Addison Wesley.

Brinch Hansen, P. (1999, April). Java's insecure parallelism. *ACM SIGPLAN 34*(4), 38–45.

Brooks, F. P. (1975). *The Mythical Man-Month.* Addison Wesley.

Brooks, F. P. (1987, April). No silver bullet: Essence and accidents for software engineering. *IEEE Computer 20*(4), 10–18.

Brooks, F. P. (1995). *The Mythical Man-Month* (Anniversary ed.). Addison Wesley.

Budd, T. A. (1995). *Multiparadigm Programming in Leda.* Addison Wesley.

Church, A. (1941). The calculi of lambda-conversion. In *Annals of Mathematics Studies*, Volume 6. Princeton Univesity Press.

Clarke, L. A., J. C. Wilden, and A. L. Wolf (1980). Nesting in Ada programs is for the birds. In *Proc. ACM SIGPLAN Symposium on the Ada programming Language.* Reprinted as ACM SIGPLAN Notices, **15**(11):139–145, November 1980.

Colmerauer, A., H. Kanoui, P. Roussel, and R. Pasero (1973). Un systeme de communication homme-machine en francais. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille.

Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare (1972). *Structured Programming.* Academic Press.

Dijkstra, E. W. (1968, August). Go to statement considered harmful. *Communications of the ACM 11*(8), 538.

Gelernter, D. and S. Jagannathan (1990). *Programming Linguistics*. MIT Press.

Gill, S. (1954). Getting programmes right. In *International Symposium on Automatic Digital Computing*. National Physical Laboratory. Reprinted as pp. 289–292 of (Williams and Campbell-Kelly 1989).

Griswold, R. E. and M. T. Griswold (1983). *The Icon Programming Language*. Prentice Hall.

Grogono, P. (1991a, January). The Dee report. Technical Report OOP–91–2, Department of Computer Science, Concordia University. ttp://www.cs.concordia.ca/∼faculty/grogono/dee.html.

Grogono, P. (1991b, January). Issues in the design of an object oriented programming language. *Structured Programming 12*(1), 1–15. Available as `www.cs.concordia/~faculty/grogono/oopissue.ps`.

Grogono, P. and P. Chalin (1994, May). Copying, sharing, and aliasing. In *Colloquium on Object Orientation in Databases and Software Engineering (ACFAS'94)*, Montreal, Quebec. Available as `www.cs.concordia/~faculty/grogono/copying.ps`.

Hanson, D. R. (1981, August). Is block structure necessary? *Software: Practice and Experience 11*(8), 853–866.

Harbison, S. P. and G. L. Steele, Jr. (1987). *C: A Reference Manual* (second ed.). Prentice Hall.

Hoare, C. A. R. (1968). Record handling. In *Programming Languages*, pp. 291–347. Academic Press.

Hoare, C. A. R. (1974). Hints on programming language design. In C. Bunyan (Ed.), *Computer Systems Reliability*, Volume 20 of *State of the Art Report*, pp. 505–34. Pergamon/Infotech. Reprinted as pages 193–216 of (Hoare 1989).

Hoare, C. A. R. (1975, June). Recursive data structures. *Int. J. Computer and Information Science 4*(2), 105–32. Reprinted as pages 217–243 of (Hoare 1989).

Hoare, C. A. R. (1989). *Essays in Computing Science*. Prentice Hall. Edited by C. B. Jones.

Jensen, K. and N. Wirth (1976). *Pascal: User Manual and Report*. Springer-Verlag.

Joyner, I. (1992). C++? a critique of C++. Available from ian@syacus.acus.oz.au. Second Edition.

Kay, A. C. (1996). The early history of Smalltalk. In *History of Programming Languages–II*, pp. 511–578. ACM Press (Addison Wesley).

Kleene, S. C. (1936). General recursive functinos of natural numbesr. *Mathematische Annalen 112*, 727–742.

Knuth, D. E. (1974). Structured programming with GOTO statements. *ACM Computing Surveys 6*(4), 261–301.

Kölling, M. (1999). *The Design of an Object-Oriented Environment and Language for Teaching.* Ph. D. thesis, Basser Department of Computer Science, University of Sydney.

Kowalski, R. A. (1974). Predicate logic as a programming language. In *Information processing 74*, pp. 569–574. North Holland.

LaLonde, W. and J. Pugh (1995, March/April). Complexity in C++: a Smalltalk perspective. *Journal of Object-Oriented Programming 8*(1), 49–56.

Landin, P. J. (1965, August). A correspondence between Algol 60 and Church's lambda-notation. *Communications of the ACM 8*, 89–101 and 158–165.

Larose, M. and M. Feeley (1999). A compacting incrmental collector and its performce in a production quality compiler. In *ACM SIGPLAN International Symposiuum on Memory Management (ISMM'98)*. Reprinted as *ACM SIGPLAN Notices* **34**3:1–9, March 1999.

Lientz, B. P. and E. B. Swanson (1980). *Software Maintenance Management.* Addison Wesley.

Lindsey, C. H. (1996). A history of Algol 68. In *History of Programming Languages*, pp. 27–83. ACM Press (Addison Wesley).

Liskov, B. (1996). A history of CLU. In *History of Programming Languages–II*, pp. 471–496. ACM Press (Addison Wesley).

Liskov, B. and J. Guttag (1986). *Abstraction and Specification in Program Development.* MIT Press.

Liskov, B. and S. Zilles (1974, April). Programming with abstract data types. *ACM SIGPLAN Notices 9*(4), 50–9.

MacLennan, B. J. (1983, December). Values and objects in programming languages. *ACM SIGPLAN Notices 17*(12), 70–79.

Martin, J. J. (1986). *Data Types and Data Structures.* Prentice Hall International.

McAloon, K. and C. Tretkoff (1995). 2LP: Linear programming and logic programming. In P. V. Hentenryck and V. Saraswat (Eds.), *Principles and Preactice of Constraint Programming*, pp. 101–116. MIT Press.

McCarthy, J. (1960, April). Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM 3*(4), 184–195.

McCarthy, J. (1978). History of LISP. In R. L. Wexelblat (Ed.), *History of Programming Languages*, pp. 173–197. Academic Press.

McKee, J. R. (1984). Maintenance as a function of design. In *Proc. AFIPS National Computer Conference*, pp. 187–93.

Meyer, B. (1988). *Object-oriented Software Construction.* Prentice Hall International. Second Edition, 1997.

Meyer, B. (1992). *Eiffel: the Language.* Prentice Hall International.

Milne, R. E. and C. Strachey (1976). *A Theory of Programming Language Semantics.* John Wiley.

Milner, B. and M. Tofte (1991). *Commentary on Standard ML.* MIT Press.

Milner, B., M. Tofte, and R. Harper (1990). *The Definition of Standard ML.* MIT Press.

Mutch, E. N. and S. Gill (1954). Conversion routines. In *International Symposium on Automatic Digital Computing.* National Physical Laboratory. Reprinted as pp. 283–289 of (Williams and Campbell-Kelly 1989).

Naur, P. (1978). The European side of the last phase of the development of Algol. In *History of Programming Languages.* ACM Press. Reprinted as pages 92–137 of (Wexelblat 1981).

Naur, P. et al. (1960, May). Report on the algorithmic language Algol 60. *Communications of the ACM 3*(5), 299–314.

Nygaard, K. and O.-J. Dahl (1978). The development of the SIMULA languages. In R. L. Wexelblat (Ed.), *History of Programming Languages*, pp. 439–478. Academic Press.

Parnas, D. L. (1972, December). On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*(12), 1053–1058.

Perlis, A. J. (1978). The American side of the development of Algol. In *History of Programming Languages*. ACM Press. Reprinted as pages 75–91 of (Wexelblat 1981).

Radin, G. (1978). The early history and characteristics of PL/I. In *History of Programming Languages*. ACM Press. Reprinted as pages 551–574 of (Wexelblat 1981).

Ritchie, D. M. (1996). The development of the C programming language. In *History of Programming Languages*, pp. 671–686. ACM Press (Addison Wesley).

Robinson, A. J. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*, 23–41.

Sakkinen, M. (1988). On the darker side of C++. In S. Gjessing and K. Nygaard (Eds.), *Proceedings of the 1988 European Conference of Object Oriented Programming*, pp. 162–176. Springer LNCS 322.

Sakkinen, M. (1992). The darker side of C++ revisited. *Structured Programming 13*.

Sammet, J. (1969). *Programming Languages: History and Fundamentals*. Prentice Hall.

Sammett, J. E. (1978). The early history of COBOL. In *History of Programming Languages*. ACM Press. Reprinted as pages 199–241 of (Wexelblat 1981).

Schwartz, J. T., R. B. K. Dewar, E. Dubinsky, and E. Schonberg (1986). *Programming with sets — an introduction to SETL*. Springer Verlag.

Steele, Jr., G. L. (1996). The evolution of LISP. In *History of Programming Languages*, pp. 233–308. ACM Press (Addison Wesley).

Steele, Jr., G. L. et al. (1990). *Common LISP: the Language* (second ed.). Digital Press.

Steele, Jr., G. L. and G. J. Sussman (1975). Scheme: an interpreter for the extended lambda calculus. Technical Report Memo 349, MIT Artificial Intelligence Laboratory.

Strachey, C. (1966). Towards a formal semantics. In T. B. Steel (Ed.), *Formal Language Description Languages for Computer Programming*. North-Holland.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison Wesley.

Taivalsaari, A. (1993). *A Critical View of Inheritance and Reusability in Object-oriented Programming*. Ph. D. thesis, University of Jyväskylä.

Teale, S. (1993). *C++ IOStreams Handbook*. Addison Wesley. Reprinted with corrections, February 1994.

Turing, A. M. (1936). On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc. 2*(42), 230–265.

Turner, D. (1976). The SASL Langauge Manual. Technical report, University of St. Andrews.

Turner, D. (1979). A new implementation technique for applicative languages. *Software: Practice & Experience 9*, 31–49.

Turner, D. (1981). KRC language manual. Technical report, University of Kent.

Turner, D. (1985, September). Miranda: a non-strict functional language with polymorphic types. In *Proceedings of the Second International Conference on Functional Programming Languages and Computer Architecture*. Springer LNCS 301.

van Wijngaarden, A. et al. (1975). Revised report on the algorithmic language Algol 68. *Acta Informatica 5*, 1–236. (In three parts.).

Wexelblat, R. L. (Ed.) (1981). *History of Programming Languages.* Academic Press. From the ACM SIGPLAN History of programming Languages Conference, 1–3 June 1978.

Wheeler, D. J. (1951). Planning the use of a paper library. In *Report of a Conference on High Speed Automatic Calculating-Engines.* University Mathematical Laboratory, Cambridge. Reprinted as pp. 42–44 of (Williams and Campbell-Kelly 1989).

Whitaker, W. A. (1996). Ada — the project. In *History of Programming Languages*, pp. 173–228. ACM Press (Addison Wesley).

Williams, M. R. and M. Campbell-Kelly (Eds.) (1989). *The Early British Computer Conferences*, Volume 14 of *The Charles Babbage Institute Reprint Series for the History of Computing.* The MIT Press and Tomash Publishers.

Wirth, N. (1971, April). Program development by stepwise refinement. *Communications of the ACM 14*(4), 221–227.

Wirth, N. (1976). *Algorithms + Data Structures = Programs.* Prentice Hall.

Wirth, N. (1982). *Programming in Modula-2.* Springer-Verlag.

Wirth, N. (1996). Recollections about the development of Pascal. In *History of Programming Languages*, pp. 97–110. ACM Press (Addison Wesley).