Phenomenology of programming languages

Programming language is used for solving the problems so it can be taken as a tool. Some of the phenomena of programming language as tool from the investigations of Don Ihde are:

- ***Tools are Ampliative and Reductive***

To better understand the phenomenology of programming languages, we may begin with a simpler tool. Ihde contrasts the experience of using your hands to pick fruit with that of using a stick to knock the fruit down. On the one hand, the stick is ampliative: it extends your reach to otherwise inaccessible fruit. On the other hand, it is reductive: your experience of the fruit is mediated by the stick, for you do not have the direct experience of grasping the fruit and tugging it off the branch. You cannot feel if the fruit is ripe before you pick it.

"Technological Utopians" tend to focus on the ampliative aspect- the increased reach and power- and to ignore the reductive aspect, whereas "technological dystopian" tend to focus on the reductive aspect- the loss of direct, sensual experience - and to diminish the practical advantages of the tool.

- ***Fascination and fear are common to new tools***

When first introduced, programming languages elicited the two typical responses to a new technology: fascination and fear. Utopians tend to become fascinated with the ampliative aspects of new tools, so they embrace the new technology and are eager to use it and to promote it (even where its use is inappropriate); they are also inclined to extrapolation: extending the technology toward further amplification. Dystopian, in contrast, fear the reductive aspects of the tool (so higher level language are fear for their efficiency), or sometimes the ampliative aspects, which may seem dangerous. Ideally, greater familiarity with a technology allows us to grow beyond these reactions.

- ***With mastery, objectification become Embodiment***

A tool replaces immediate (direct) experience with mediated (indirect) experience. Yet, when a good tool is mastered, its mediation becomes transparent. Consider again the stick. If it is a good tool (sufficiently stiff, not too heavy, etc.) and if you know how to use it, then it functions as an extension of your arm, allowing you both to feel the fruit and to act on it. In this way the tool becomes partially embodied. On

the other hand, if the stick is unsuitable or you are unskilled in its use, then you experience it as an object separate from your body; you relate to it rather than through it. With mastery a good tool becomes transparent; it is not invisible, for we still experience its ampliative and reductive aspects, but we are able to look through it rather than at it.
As you acquire skill with the language, it becomes transparent so that you can program the machine through the language and concentrate on the project rather than the tool. With mastery, objectification yields (partial) embodiment.

- ***Programming Language influence focus and action***

Tools influence the style of a project. E.g. writing technologies: dip pen, an electric typewriter, and a word processor. In case of dip pen it is slower than the speed of thought, with typewriter the speed is closer to the speed of thought, and with word processor, text can be revised and rearranged in small units, so there is greater tendency to salvage bits of text.
In general, a tool influences focus and action. It influences focus by making some aspects of the situation salient and by hiding others. Like others, programming language influence the focus and actions of programmers and therefore their programming style.

Programming Language

A programming language is the collection of syntactic rules, keywords, naming structures, data structures, expression and control structures that is intended for the expression of computer programs and that is capable of expressing any computer program.

Programming Language is a formal method that:

- Describe a solution to a problem
- Organize a solution to a problem
- Reason about a solution to a problem
- Interface between user and machine Programming languages trade-off:

- Ease of use - high level
- Efficiency - low level

"A programming language is a language that is intended for the expression of computer programs and that is capable of expressing any computer program."

Characteristics of a Good Programming Languages

1. Clarity, Simplicity and Unity

A programming language provides a medium to conceptual thinking of new algorithms and also a medium to execute your thought process into real coding statements. For algorithms to be implemented on a language it's a basic need is that the language is quite *clear, simple and unified* in structure. Such that the Primitives of language can be utilized to develop algorithms. It is desirable to have a minimum number of different concepts, so that combining multiple concepts won't be that complex in nature. It should be simple and regular as possible. This attribute of a language is known as conceptual Integrity.

The main concern of a language now a days is its readability. The syntax of language effects the ease with which programs are written, tested and later used for knowledge or research purpose. A complex syntax language may be easy to write program in, but it proves to be difficult to read and debug the code for later sessions. **For example** APL programs are so complicated that even the own developers find it difficult to understand after 1-2 months. The language should be simple enough to understand or point out errors.

2. Orthogonality

The term orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. Language design must follow orthogonality principle i.e. independent functions should be controlled by independent mechanisms **For example,** suppose a language provides with an expression let's say an arithmetical calculation operator .Taking Another Expression facilitated by the language like conditional Statement, which has 2 outputs either 0 or 1 (in some cases TRUE or FALSE). Now the language should support combination of these two expressions. So that new statement can be formed, and this orthogonality helps to develop many new algorithms.

3. Naturalness for the application

A language needs a Syntax that, when applied properly, allows the program Structure to reflect the logical structure what a programmer wanted it to. *Arithmetic*

Algorithms ,concurrent algorithms , logic Algorithms and other type of statement have differing natural structures , that can be represented by the Program in that language. The language should provide appropriate *data structures, operations, control structures and a natural syntax* for the problem to be solved.

For Example: Consider a real life condition of plates being placed above each plate, this structure is known as Stack. This Stack can be implemented into programming world also. This is used as a data Structure in most of the Languages.

4. Support for Abstraction

Many times languages fail to implement many real life problems into Programs. There is always a gap between abstract data structures and operations. Even most natural Programming language fails to bridge the gap. **For Example:** Consider a situation where a scheduling is to be done for college *student for attending a lecture in a class section, teacher.* Suppose the requirement is to assign a student a section lecture and teacher to attend, which are common task for natural application, but are not provided by C.

The need of point is to design an appropriate abstraction for the problems solution and then implementing these abstraction using most primitive features of a language. Ideally, the language should provide the data structures, data types and operations to maintain such abstractions .C++ is one of the most used language, that provide such facilities.

5. Ease of Program Verification

The reliability of a programming Language written in a language is always a central Concern. There are many techniques which can be used to keep track of correct functionality of a language. Sometimes testing the Program with random values of the inputs and obtaining corresponding outputs. Program verification should be provided by languages to check and minimize the errors.

6. Programming Environment

The environment also plays a vital role in success of a Language. The environment which is technically weak, may get a bad response of Programmer, rather than a language that has less facility than the former but its environment is Technically Good. Some of the Good featured of an environment are Special editors and testing packages tailored to the language may greatly speed up the creation and testing of Programs.

7. Portability of Programs

The important criterion for many programming projects is the Transportability of the resulting program from one computer to another systems. A language which is widely available and do not support different features on different computer System, which may have different hardware, is considered a good language. **For Example** *C, C++* and most of the language now days are Portable in nature.

8. Cost of Use

The trickiest point that always matter a lot in any system that uses resources. It's a major element to decide the Evaluation of any programming language, but cost means many different things.

(a) Cost of Program Execution

Program Execution cost is total amount which has been used to implement the program. The research work on design, optimizing compilers, data allocation registers etc. These are the basic things which come under the cost of Program Execution.

(b) Cost of Program Translation

The next concern is program compilation. The program is compiled many times than it is being executed. In such case, it is important to have a speed and efficient compiler to handle this Job.

(c) Cost of Program Creation, Testing and Use

Another aspect of Cost management. This includes the cost which a programmer charges for his work of creating Project with the Specified features, the cost involving the Testing issues.

(d) Cost of Program Maintenance

After a program is being installed in a System, then after certain intervals it needs maintenance to run smoothly. The maintenance includes the rectification of Error propagated in real time, the updating of Program as need of time.

Complexity in programming and understanding programs led to development of program design notations. Explain with the reference to pseudocode.

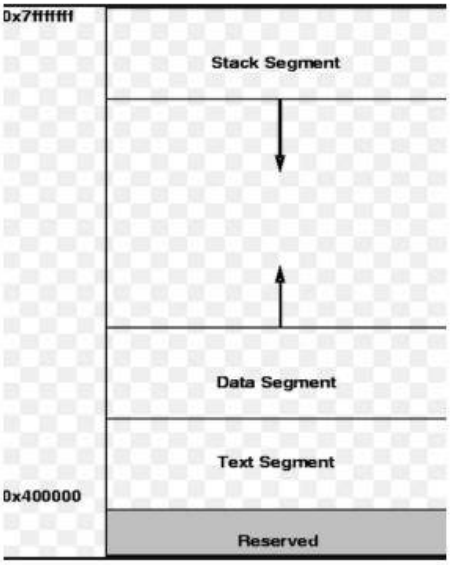
Design Notations

Complexity in programming and understanding programs led to development of *program design notations*. These were designed to help the programmer, not to be interpreted by computers.

The notation for doing this may be called an *abstract programming language* because programs written in the notation are not intended to be run directly on a computer, they have to be coded into a real programming language. An abstract programming language performs the same role that in earlier times flowcharting was supposed to do; it is an informal notation to be used *during* program design.

Some of these notations helped the programmer to design the:

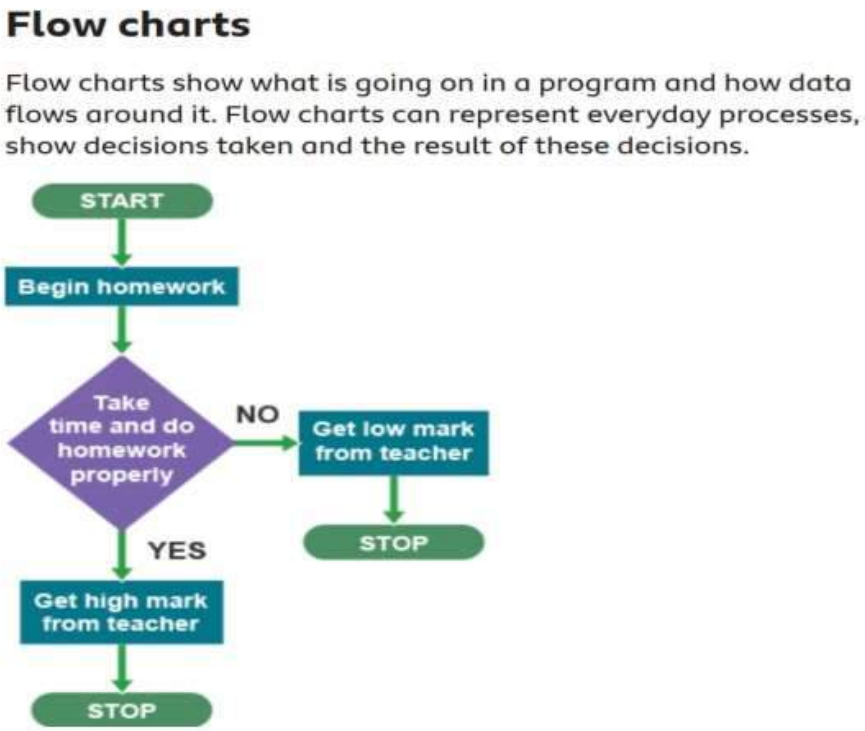
Memory layout



Pseudo-code

Control flow

- *Flow Diagrams*
- Later: Flowcharts

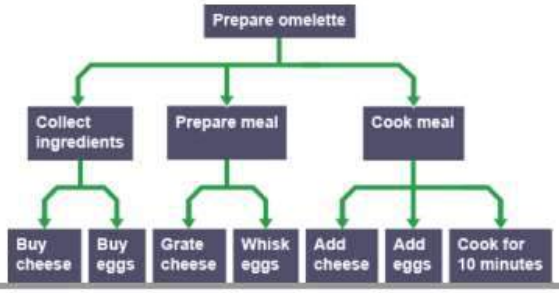


Mnemonics

- Helps to remember instruction codes like assembly language today

Structure Diagram

Another way of representing a program design is to use a structure diagram. Structure diagrams break down a problem into smaller sections. These smaller sections can then be worked on one at a time.



PseudoCode

It is an instruction code that is different than that provided by the real machine. Pseudocode offered floating point support and indexing which was actually not supported by earlier generation computers. It provided entirely new instruction set not offered by the real hardware.

Need of PseudoCode

During first generation of computer, programming was very difficult as the programmer need to know about the hardware specification of every machine that the program was intended for. For example in 1950's for IBM 650 which has following characteristics:

- No programming language was available (not even assembler)
- Memory was only a few thousand words.
- Stored program and data on rotating drum.
- Instructions included address of next instruction so that rotating drum was under next instruction to execute and no full rotations were wasted.

PseudoCode Interpreters

Pseudo-Code Interpreter is an interpretive subroutine (The subroutine is an important part of any computer system's architecture. A subroutine is a group of instructions that usually performs one task, it is a reusable section of the software that is stored in memory once, but used as often as necessary.) developed to run the pseudocode. They are used for saving memory since the pseudocode is more compact than machines real program code. It implements a virtual computer which allows us to use functionality with its own data types (e.g. floating point) and operations (e.g. indexing) not provided by the actual hardware in which the virtual machine resides but abstracted by the virtual machine. It has own data types and operations and we can view all programming languages this way .

The pseudocode was at the higher level and provided facilities more suitable to applications and it eliminated many details from programing. In general it is an example of "Automation Principle" of programming language.

Pseudo-Code interpreters were commonly used to perform floating-point operations and indexing. Consistent use of these simplified the programming process and this

simulated instructions not provided by the hardware.
Pseudo-Code interpreter (a primitive, interpreted programming language) implements:
 A virtual computer
 New instruction set
 New data structures
Virtual computer:
 Higher level than actual hardware
 Provides facilities more suitable to applications
 Abstracts away hardware details

PseudoCode follows two basic Principles of Programming
The Automation Principle
Automate mechanical, tedious, or error prone activities.
The Regularity Principle
Regular rules, without exceptions, are easier to learn, use, describe, and implement.

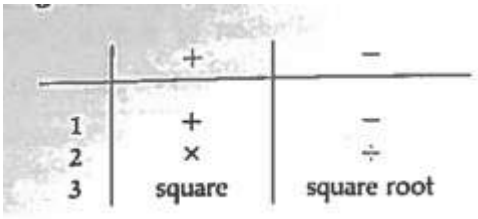
Design of a Pseudo-Code
The design of PseudoCode is based on the capabilities and constraints of the first generation computers. In 1950 Capabilities expected by the programmers and not support by the hardware at that time are:
 Floating point operation support (+,-,*,/,…)
 Comparisons (=,≠,<,≤,>,≥)
 Indexing
 Transfer of control
 Input/output

Hardware Assumptions
The IBM 650 will serve as the hardware
 1 word: 10 decimal digits + 1 sign
 2000 byte memory
- 1000 for data
- 1000 for program

The design of pseudocode must follow impossible error principle because " making errors impossible to commit is preferable to detecting them after their commission". E.g.: Cannot modify the program accidentally, since memory modifying operations are for "data memory" only.

Artihmetic operation in pseudocode
Since arithmetic operation are the most common one, let's first consider this:
Two address: x+y → x
Three address: x+y→z requires nine digit
Four address: x+y*z→w requires 12 digit, not suitable in 10-digit word.

operation val1 val2 destination
+1 110 120 200
+, -, /, x, square, square root



	+	−
1	+	−
2	×	÷
3	square	square root

Comparisons

We have said that we want our virtual computer to be regular so that it will be easier to use than the real computer. Achieving regularity will be easier if we use the same format for our other operations that we've used for the arithmetic operations. Let's see how this applies to the comparison operations. In some sense *equal* is the inverse of *not equal*, and *greater than or equal* is the inverse of *less than*, so we can use signs to distinguish between each operation and its inverse. We extend the operation table as follows:

	+	−
1	+	−
2	×	÷
3	square	square root
4	if = goto	if ≠ goto
5	if ⩾ goto	if < goto

For example, the instruction

+4 200 201 035

means: If the contents of (data) location 200 equal the contents of (data) location 201, then goto the instruction in (program) location 035. Notice that it is not

Indexing

One of the justifications for our pseudo-code was that it provided built-in indexing, so we will turn to the design of this facility next. To perform indexing we will need the address of the array and the address of the index variable, thus consuming two of the three address fields in the instruction. Therefore, the only operations we can perform directly on array elements are to move them to or from other locations. We can use the codes +6 and −6 to move from or to an array: $x \leftarrow z$ and $x \rightarrow y$. The formats of these operations are:

+6 *xxx iii zzz*
−6 *xxx yyy iii*

For example, if there is a 100-element array beginning at location 250 in data memory, and location 050 contains 17, then

+6 250 050 803

will move the contents of location 267 (= 250 + 17) to location 803. Similarly,

−6 722 250 050

will move the contents of location 722 to location 267.

Input-Output

The only functions in our list that we have not yet addressed are the input and output operations. A program is not usually useful if it can't read data or print a result. Therefore, we will use the +8 operation to read a card containing one 10-digit number into a specified memory location and the −8 operation to print the contents of a memory location. (In a real pseudo-code, a punch operation would be more common than a print operation since this would allow the output of one program to be used as the input to another.) The complete list of operations is summarized in Figure 1.2. Notice that we have added a stop instruction to terminate program execution.

Programming Paradigm

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

Common programming paradigms include:[1][2][3]

- imperative in which the programmer instructs the machine how to change its state,
  - procedural which groups instructions into procedures,
  - object-oriented which groups instructions together with the part of the state they operate on,
- declarative in which the programmer merely declares properties of the desired result, but not how to compute it
  - functional in which the desired result is declared as the value of a series of function applications,
  - logic in which the desired result is declared as the answer to a question about a system of facts and rules,
  - mathematical in which the desired result is declared as the solution of an optimization problem

Symbolic techniques such as reflection, which allow the program to refer to itself, might also be considered as a programming paradigm. However, this is compatible with the major paradigms and thus is not a real paradigm in its own right.

Imperative Programming

The imperative programming paradigm is based on the Von Neumann architecture of computers, introduced in 1940's. Von Neumann architecture is the dominant computer hardware architecture which consists of a single sequential CPU separate from memory, and with data piped between CPU and memory. This is reflected in the design of the imperative languages, with

• States — representing memory cells with changing values,
• Sequential orders — reflecting the single sequential CPU, and
• Assignment statements — reflecting piping.

Imperative programs are sequences of directions (or orders) for performing an action. Therefore, imperative programming is characterized by programming with states and commands which modify these states. Imperative programming languages provide a variety of commands in order to structure the code and to manipulate the states. Usually, in imperative programming languages, a sequence of commands can be named and the name can be used to invoke the sequence of commands. Named sequence of commands is called subprogram, procedure or function. When

imperative programming is combined with subprograms it is called procedural programming.

Imperative paradigm is supported by languages such as FORTRAN (introduced in 1954), Cobol (1959), Pascal (1970), C (1971), and Ada (1979), . . .

Object-Oriented Programming:

The object-oriented programming is a generalization of imperative programming. The conceptual model of this paradigm is developed from simulation of events. The main underlying idea of this model is: the structure of the simulation should reflect the environment that is being simulated. If real world phenomena are simulated, then there should be an object for each entity involved in the phenomena. Object is an entity encapsulating data and related operations. As in the real world, objects interact—so, object-oriented programming uses message passing to capture interactions between objects.

A programming language supporting this concept and using objects is called object-based. Object-oriented programming languages support additional features, with the following most important ones:

• Abstract data type definitions are used to define properties of classes of objects;

• Inheritance is a mechanism that allows definition of one abstract data type by deriving it from an existing abstract data type—the newly defined type inherits the properties of the parent type;

• Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes;

• Dynamic binding of function calls supports the use of polymorphic functions; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable.

Object-oriented paradigm is supported by languages such as Smalltalk (1969), C++ (1983), and Java (1995).

Functional Programming:

The functional programming paradigm is based on the theory of mathematical functions, more precisely on the lambda-calculus. It allows the programmer to think about the problem at a higher level of abstraction—it encourages thinking about the nature of the problem rather than about sequential nature of the underlying computing engine. Functional languages are motivated and developed by the

following questions: what is the proper unit of program decomposition and how can a language best support program composition from independent components.
A functional programming language usually has three main sets of components:
• data objects — such as a list or an array;
• built-in functions — for manipulating the basic data objects;
• functional forms — also called high-order functions, for building new functions (such as composition and reduction).
Functional programming languages are called applicative since the functions are applied to their arguments, and non-procedural or declarative since the definitions specify what is computed and not how it is computed.
Functional paradigm is supported by languages such as LISP (1958), ML (1973), Scheme (1975), Miranda (1982), and Haskell (1987).

Logic Programming

The logic programming paradigm is based on first-order predicate calculus. This programming style emphasizes the declarative description of a problem rather than the decomposition of the problem into an algorithmic implementation. A logic program is a collection of logical declarations describing the problem to be solved. As such, logic programs are close to specifications. The problem description is used by an inference engine to find a solution. More precisely, a logic program consists of:
• axioms — defining facts about objects,
• rules — defining ways for inferencing new facts,
• a goal statement — defining a theorem, potentially provable by given axioms and rules.
Logic programming is characterized by programming with relations and inference. The programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Logic languages are usually more demanding in computational resources than procedural and object-oriented languages. Logic paradigm is supported by languages such as Prolog (1970), and G¨odel (1994). Curry (1997) is a multiparadigm programming language merging elements of functional and logic programming.

Reasons for Studying Principles of Programming Languages

Reasons for Studying Principles of Programming Languages

It is natural for students to wonder how they will benefit from the study of programming language concepts. After all, many other topics in computer science are worthy of serious study. The following is what we believe to be a compelling list of potential benefits of studying concepts of programming languages:

Increased capacity to express ideas.

It is widely believed that the depth at which people can think is influenced by the expressive power of the language in which they communicate their thoughts.

Programmers, in the process of developing software, are similarly constrained. The language in which they develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, the forms of algorithms they can construct are likewise limited. Awareness of a wider variety of programming language features can reduce such limitations in software development. Programmers can increase the range of their software development thought processes by learning new language constructs.

Improved background for choosing appropriate languages

- *Improved background for choosing appropriate languages.* Some professional programmers have had little formal education in computer science; rather, they have developed their programming skills independently or through in-house training programs. Such training programs often limit instruction to one or two languages that are directly relevant to the current projects of the organization. Other programmers received their formal training years ago. The languages they learned then are no longer used, and many features now available in programming languages were not widely known at the time. The result is that many programmers, when given a choice of languages for a new project, use the language with which they are most familiar, even if it is poorly suited for the project at hand. If these programmers were familiar with a wider range of languages and language constructs, they would be better able to choose the language with the features that best address the problem.

  Some of the features of one language often can be simulated in another language. However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe.

Increased ability to learn new languages.

Computer programming is still a relatively young discipline, and design methodologies, software development tools, and programming languages are still in a state of continuous evolution. This makes software development an exciting profession, but it also means that continuous learning is essential. The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only one or two languages and has never examined programming language concepts in general. Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.

Better understanding of the significance of implementation.

- *Better understanding of the significance of implementation.* In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts. In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are. In turn, this knowledge leads to the ability to use a language more intelligently, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices.

  Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. Another benefit of understanding implementation issues is that it allows us to visualize how a computer executes various language constructs. In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program. For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called can be a highly inefficient design choice.

- *Better use of languages that are already known.* Most contemporary programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses. By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.
- *Overall advancement of computing.* Finally, there is a global view of computing that can justify the study of programming language concepts. Although it is usually possible to determine why a particular programming language became popular, many believe, at least in retrospect, that the most popular languages are not always the best available. In some cases, it might be concluded that a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts.

Basic Principles of Programming Language

Abstraction

Avoid requiring something to be stated more than once; factor out the recurring pattern.

Automation

Automate mechanical, tedious, or error-prone activities.

Defense in Depth Principle

If an error gets through one line of defense, then it should be caught by the next line of defense.

Elegance

Confine your attention to design that look good because they are good.

Impossible Error

Making errors impossible to commit is preferable to detecting them after their commission.

Information Hiding

Modules should be designed so that

1) The users have all the information needed to use the module correctly, and nothing more.

2) The implementor has all the information needed to implement the module, correctly, and nothing more.

Labelling

Avoid arbitrary sequence more than a few items long; do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.

Localized Cost

Users should only pay for what they use; avoid distributed costs.

Orthogonality
Independent functions should be controlled by independent mechanisms.

Portability
Avoid features or facilities that are dependent on a particular computer or a small class of computers.

Regularity
Regular rules, without exceptions, are easier to learn, use, describe, and implement.

Syntactic Consistency
Things that look similar should be similar and things that look different should be different.

Zero-One-Infinity
The one reasonable number in programming language design are zero, one, and infinity.

Programming Domains

Computers have been applied to a myriad of different areas, from controlling nuclear power plants to providing video games in mobile phones. Because of this great diversity in computer use, programming languages with very different goals have been developed.

1.2.1 Scientific Applications

The first digital computers, which appeared in the late 1940s and early 1950s, were invented and used for scientific applications. Typically, the scientific applications of that time used relatively simple data structures, but required large numbers of floating-point arithmetic computations. The most common data structures were arrays and matrices; the most common control structures were counting loops and selections. The early high-level programming languages invented for scientific applications were designed to provide for those needs. Their competition was assembly language, so efficiency was a primary concern. The first language for scientific applications was Fortran. ALGOL 60 and most of its descendants were also intended to be used in this area, although they were designed to be used in related areas as well. For some scientific applications where efficiency is the primary concern, such as those that were common in the 1950s and 1960s, no subsequent language is significantly better than Fortran, which explains why Fortran is still used.

1.2.2 Business Applications

The use of computers for business applications began in the 1950s. Special computers were developed for this purpose, along with special languages. The first successful high-level language for business was COBOL (ISO/IEC, 2002), the initial version of which appeared in 1960. It probably still is the most commonly used language for these applications. Business languages are characterized by facilities for producing elaborate reports, precise ways of describing and storing decimal numbers and character data, and the ability to specify decimal arithmetic operations.

There have been few developments in business application languages outside the development and evolution of COBOL. Therefore, this book includes only limited discussions of the structures in COBOL.

1.2.3 Artificial Intelligence

Artificial intelligence (AI) is a broad area of computer applications characterized by the use of symbolic rather than numeric computations. Symbolic computation means that symbols, consisting of names rather than numbers, are manipulated. Also, symbolic computation is more conveniently done with linked lists of data rather than arrays. This kind of programming sometimes requires more flexibility than other programming domains. For example, in some AI applications the ability to create and execute code segments during execution is convenient.

The first widely used programming language developed for AI applications was the functional language Lisp (McCarthy et al., 1965), which appeared in 1959. Most AI applications developed prior to 1990 were written in Lisp or one of its close relatives. During the early 1970s, however, an alternative approach to some of these applications appeared—logic programming using the Prolog (Clocksin and Mellish, 2013) language. More recently, some AI applications have been written in systems languages such as C. Scheme (Dybvig, 2009), a dialect of Lisp, and Prolog are introduced in Chapters 15 and 16, respectively.

1.2.4 Web Software

The World Wide Web is supported by an eclectic collection of languages, ranging from markup languages, such as HTML, which is not a programming language, to general-purpose programming languages, such as Java. Because of the pervasive need for dynamic Web content, some computation capability is often included in the technology of content presentation. This functionality can be provided by embedding programming code in an HTML document. Such code is often in the form of a scripting language, such as JavaScript or PHP (Tatroe, 2013). There are also some markup-like languages that have been extended to include constructs that control document processing, which are discussed in Section 1.5 and in Chapter 2.

Fortran

Name Structure in Fortran

As data structure organize the Data, control structure organizes the control flow of program, name structure organizes the names that appear in a program.

When we declare a new variable in a program:

```
INTEGER I
```

It bound the identifier I to a location in a memory. Here INTEGER specify the type of the variable, and I is the identifier.

Declarative constructs are the primitives name structure of FORTRAN.

In our pseudo-code we saw that the declarations performed three functions:

1. They *allocated* an area of memory of a specified size.
2. They attached a symbolic name to that area of memory. This is called *binding* a name to an area of memory.
3. They *initialized* the contents of that memory area.

These are three important functions of declarations in most languages, including FORTRAN. For example, the declaration

```
DIMENSION DTA (900)
```

causes the loader to allocate 900 words and to bind the name "DTA" to this area. A separate kind of declaration, called a DATA declaration, can be used for initialization. For example,

```
DATA DTA, SUM / 900*0.0, 0.0
```

would initialize the array DTA to 900 zeroes and the variable SUM to zero. FORTRAN does not require the programmer to initialize storage; this lack of initialization is a frequent cause of errors. Declarations and initializations are discussed in Section 2.3.

Example
INTEGER I J K
Here storage must be allocated to variables I, J, K and the names I, J, K must be bound to the address of these location.

It will make entries in the symbol table for I, J, K. These entries will consist of location of these variables and their type.

Name	Type	Location
⋮	⋮	⋮
I	INTEGER	0245
J	INTEGER	0246
K	INTEGER	0247
⋮	⋮	⋮

Optional Variable Declarations Are Dangerous

FORTRAN has an unusual convention that has been abandoned in almost all newer programming languages: automatic declaration of variables. In other

words, if a programmer uses a variable but never declares it, the declaration will be automatically supplied by the compiler. This was originally conceived as a convenience for programmers since it saved them the effort of declaring all of their variables. As we will see below, it is a false economy since it undermines security.

You may ask, "If FORTRAN automatically declares undeclared variables, then how does it know what type the variable should be declared to be?" In other words, if the statement 'I = I + 1' appears in a program, and I has not been declared, then does the compiler declare it INTEGER, REAL, COMPLEX, or what? FORTRAN solves this problem with the "I Through N Rule," which states that any variables whose names begin with I through N are declared integers and all others are declared reals. This tends to correspond to mathematical convention since the variables *i*, *j*, *k*, *l*, *m*, and *n* are usually used for indexes and counts (i.e., integers), and the variables *x*, *y*, *z*, *a*, *b*, *c*, and so forth, are usually used for other things. This rule has the unfortunate consequence that it leads programmers to pick obscure names for variables (e.g., KOUNT, ISUM, XLENGTH) so that they don't have to declare them.

There is a much more serious problem with this automatic declaration convention, however. Suppose a programmer intended to type the statement 'COUNT = COUNT + 1' but accidently typed 'COUNT = COUMT + 1' (i.e., an 'M' instead of an 'N'). What would be its effect? Since the variable COUMT has not been declared, the compiler will automatically declare it, as a real in this case. Since there is no DATA-statement to initialize it, it will probably contain whatever was left over in that memory location. The result will be that a strange and inexplicable value will be stored into COUNT and that the programmer will have a difficult debugging problem. If FORTRAN did not automatically declare variables, this error would have been caught at compile-time, since

History of FORTRAN

Now: FORTRAN The First Generation

- Early 1950s
  - Simple assemblers and libraries of subroutines were tools of the day
  - Automatic programming was considered unfeasible
  - Good coders liked being masters of the trade
- Laning and Zierler at MIT in 1952
  - Algebraic language

Backus at IBM

- Visionary at IBM
- Recognized need for faster coding practice
- Need "language" that allows decreasing costs to linear, in size of the program
- Speedcoding for IBM 701
  - Language based on mathematical notation
  - Interpreter to simulate floating point arithmetic

Backus at IBM

- Goals
  - Get floating point operations into hardware: IBM 704
    - Exposes deficiencies in pseudo-code
  - Decrease programming costs
    - Programmers to write in conventional mathematical notation
    - Still generate efficient code
- IBM authorizes project
  - Backus begins outlining FORTRAN
    - IBM Mathematical FORmula TRANslating System
  - Has few assistants
  - Project is overlooked (greeted with indifference and skepticism according to Dijkstra)

Meanwhile

- Grace Hopper organizes Symposia via Office of Naval Research (ONR)
- Backus meets Laning and Zierler
- Later (1978) Backus says:
  - "As far as we were aware we simply made up the language as we went along. We did not regard language design as a difficult problem, merely as a simple prelude to the real problem: designing a compiler which could produce efficient programs."
- FORTRAN compiler works!

FORTRAN Has Been Revised Several Times

The experience gained with this FORTRAN system led Backus and his colleagues to propose FORTRAN II in September 1957. The compiler was available in the spring of 1958, and the language remains in use to this day. A dialect called FORTRAN III was designed and implemented in late 1958, but it never achieved widespread use because of its many dependencies on the IBM 704. In 1962 the FORTRAN IV language was designed; it is still an important FORTRAN dialect. In *Programming Languages: History and Fundamentals*, Jean Sammet said, "By 1963 virtually all manufacturers had either delivered or committed themselves to producing some version of FORTRAN." Reflecting FORTRAN's popularity, the American National Standards Institute (ANSI) began development of standard FORTRAN IV (ANS FORTRAN) in 1962; this was completed in 1966. Unfortunately, dialects of FORTRAN are very common and few compilers implement exactly the ANSI standard. In 1977 a new ANS FORTRAN was developed that is sometimes known as FORTRAN 77. This language incorporates many ideas from later languages, which gives it a very different appearance. It may be that it has gone too far, in that it is so different from previous FORTRAN versions that it will not benefit from their popularity, and it has not gone far enough, in that it is still a 1950s language in its capabilities. In any case, we will concentrate on the 1966 ANS FORTRAN since our intent is to illustrate the characteristics of first-generation languages, and these are more apparent in FORTRAN IV. Unless otherwise specified, in this chapter the name FORTRAN refers to ANS FORTRAN IV.

1. GOTO's are depreciated and if-else-then and while-do are promoted.
2. FORTRAN 77 became American National Standard.
3. Modularization, Pointers.
4. FORTRAN 90 supports exception handling, oop.

Fortran will continue to evolve.

Dynamic chain of Activation Records

Dynamic chain is the pointer that reach back from callee's activation record to the caller activation record. It is the primary mean to implement subprograms.

How to implement subprograms?

Clearly, the very first step is to transmit the parameters to the subprogram. This may be done by pass by value or pass by reference.

The next steps seems to be enter the subprogram, but if we do that, there will be no path to get back to caller as subprograms can be called from many different callers.

There is another issue we must address: saving the state of the caller. Registers are used for high-speed temporary storage. Since the callers and calle may be separately compiled there are no way to know the content of registers. So a private area storage must be created, so that the content of registers can be stored back when te caller gets control back from the callee. This data area is often called as activation record because it holds the information relevant to the activation of subprograms.

Let's summarize the tasks that must be completed to perform a subprogram invocation.

1. Place the parameters in the callee's activation record.
2. Save the state of the caller in the caller's activation record (including the point at which the caller is to resume execution).
3. Place a pointer to the caller's activation record in the callee's activation record.

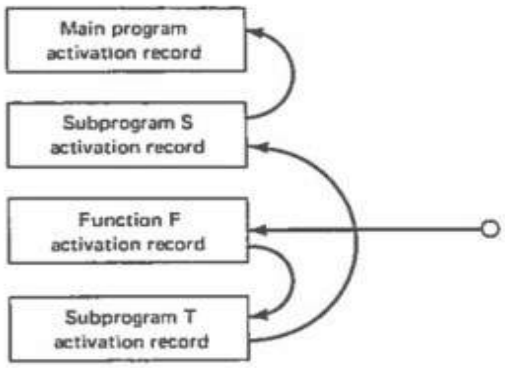


Figure 2.3 The Dynamic Chain of Activation Records

4. Enter the callee at its first instruction.

The steps required to return from the callee to the caller are as follows:

1. Get the address at which the caller is to resume execution and transfer to that location.
2. When the caller regains control, it will have to restore the rest of the state of its execution (registers, etc.) from its activation record.

In addition, if the callee were a function (as opposed to a subroutine), then the value it returns must be made accessible to the caller. This can be accomplished by leaving it in a machine register or by placing it in a location in the caller's activation record.

From this discussion, we can see that an activation record must contain space for the following information:

1. The parameters passed to this subprogram when it was last called (we call this part PAR, for parameters)
2. The IP, or resumption address, of this subprogram when it is not executing
3. The dynamic link, or pointer to the activation record of the caller of this subprogram (we call this DL)
4. Temporary areas for storing register contents and other volatile information (we call this TMP)

Control Structure of FORTRAN

Control structure are those constructs in the language that governs the flow of control of program.

Types of control structure in FORTRAN

1. Lower Level Control Structure: IF-statement, GOTO-statement
2. Higher Level Control Structure: Do-Loop

Arithmetic IF-statement

- Example of machine dependence
  - `IF (a) n1, n2, n3`
  - Evaluate a: branch to
    - n1: if -,
    - n2: if 0,
    - n3: if +
  - CAS instruction in IBM 704
- More conventional IF-statement was later introduced
  - `IF (X .EQ. A(I)) K = I - 1`

GOTO

- Workhorse of control flow in FORTRAN
- 2-way branch:

```
    IF (condition) GOTO 100
          case for false
    GOTO 200
100       case for true
200
```

- Equivalent to *if-then-else* in newer languages

Computed GOTO vs Assigned GOTO

Good: very flexible, can implement, elaborate control structures

Bad: hard to know what is intended, Violates the structure principle

Computed GOTO

GOTO (L1, L2, L3 … LN), I

The **computed** GO TO statement selects one statement label from a list, depending on the value of an integer or real expression, and transfers control to the selected one.

Example

Example: Computed GO TO

```
      ...
      GO TO ( 10, 20, 30, 40 ), N
      ...
10    CONTINUE
      ...
20    CONTINUE
      ...
40    CONTINUE
```

:

In the above example:

- If N equals one, then go to 10.
- If N equals two, then go to 20.
- If N equals three, then go to 30.
- If N equals four, then go to 40.
- If N is less than one or N is greater than four, then fall through to 10.

Assigned GOTO statement

GOTO N, (L1, L2, L3 …, LN)

This statement transfers to the statement whose address is in the variable N.

identical. Therefore, it is not uncommon for a programmer to write one where the other is expected. Let's consider the consequences of writing a computed GOTO where an assigned GOTO is intended:

```
ASSIGN 20 TO N
   :
GOTO (20, 30, 40, 50), N
```

The ASSIGN-statement will assign the address of statement number 20 (say, 347) to N. The computed GOTO will then attempt to use this as an index into the jump table (20, 30, 40, 50). In this case, the index (347) will be well out of range, but most systems don't check this so we will fetch some value out of memory to use as the destination of the jump. The result is that the program will transfer to an unpredictable location in memory, thus leading to a very-difficult-to-find bug.

Now let's consider the opposite error: using an assigned GOTO where a computed GOTO is intended.

```
I = 3
  :
GOTO I, (20, 30, 40, 50)
```

Since the assigned GOTO expects the variable I to contain the address of a statement, it will transfer to that address. In this case, it will transfer to address 3, which is almost certainly not the address of one of the statements in the list and is very likely not the address of a statement at all (low-addressed locations are often dedicated to use by the system). Again, a difficult bug results.

What are the causes of these problems? The most obvious cause is the easily confused syntax of the two constructs:

GOTO (L_1 ..., L_n), I
GOTO I, (L_1, ..., L_n)

This is a violation of the Syntactic Consistency Principle.

> ***Syntactic Consistency Principle***
>
> Things which look similar should be similar and things which look different should be different.

DO LOOP

```
      DO 100 I = 1, N
      A(I) = A(I)**2
100   CONTINUE
```

is a command to execute the statements between the DO and the corresponding CONTINUE with I taking on the values 1 through N. The variable that changes values (I in this case) is called the *controlled variable*, and the statements that are repeated, which extend from the DO to the CONTINUE with a matching label, are called the *extent* or *body* of the loop.

3. Subprogram control structures(CALL and RETURN)

Parameter Passing in FORTRAN
FORTRAN allows parameters to be used for input, output or both.

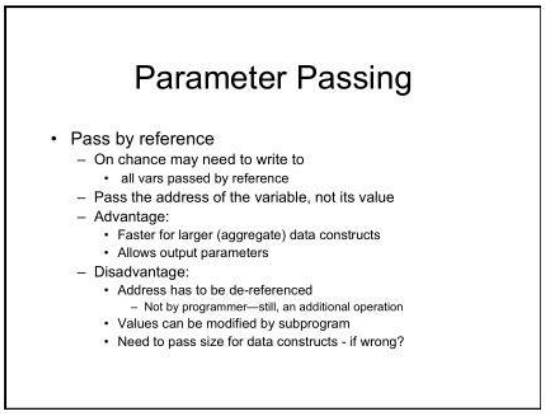
```
SUBROUTINE DIST (D, X, Y)
D = X – Y
IF (D .LT. 0) D = -D
RETURN
END
```

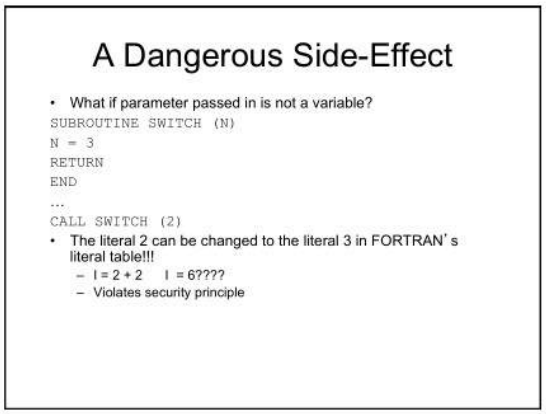
.
.

```
CALL DIST(diff1, X1, Y1)
```

.
Here, X1 and Y1 are used as input variables. Diff1 is used as output variables. The content of the diff1 is modified.

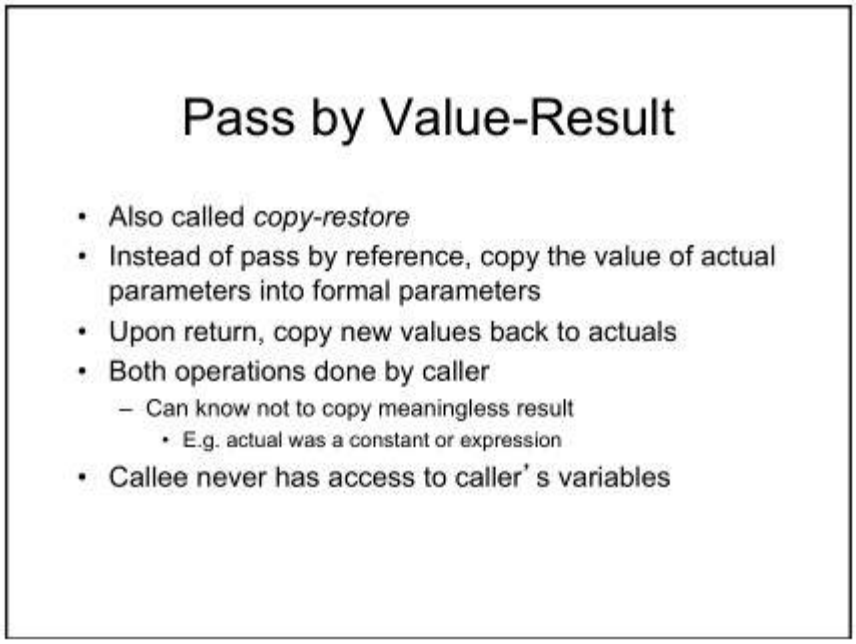
Parameters are usually passed by reference.





Although the best solution to this problem is to allow the programmer to specify which parameter is to be used for input or output.

Pass by value-result is preferable



Data Structures in FORTRAN

FORTRAN is a scientific programming language. Therefor, the data structure included in FORTRAN were those most familiar to scientific and engineering applications of mathematics: scalars and arrays.

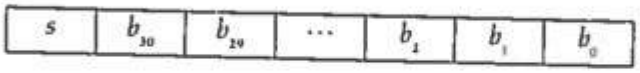
1. Scalar Data Types(Primitive)
2. Arrays Data Types

Scalar Data Types

Since scientific programming makes heavy use of numbers, it follows that numeric scalar are the primary data types in FORTRAN.

INTEGER

Each of the primitive data types was *represented* in a manner appropriate to the operations defined on that data type. For example, integers were usually represented as a binary number with a sign bit:

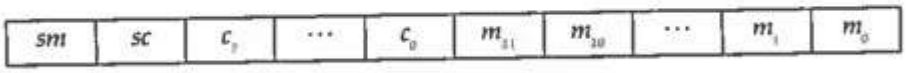


For the sake of this and the following examples, we will presume a computer with a 32-bit word. The number represented by this bit pattern is:

$$(-1)^{s} \sum_{i=0}^{30} b_i 2^i$$

Real Numbers

As you know, floating-point numbers are related to scientific notation, that is, to the convention of representing a number by a coefficient and a power of 10, for example, -1.5×10^3. A typical representation for a floating-point number is:



The value represented by this number is $m \times 2^{e}$, where m is the *mantissa*.

Array Data Structure

Arrays Are Static and Limited to Three Dimensions

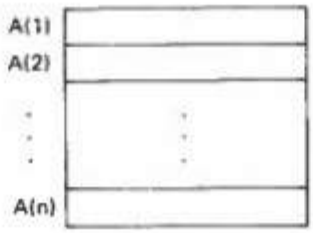
In FORTRAN arrays are declared by means of a DIMENSION statement. For example,

```
DIMENSION DTA(100), COORD(10,10)
```

declares DTA to be a 100-element array (with subscripts in the range 1–100) and COORD to be a 10 × 10 array (with subscripts in the range 1–10). We can see

DIMENSION $A(n)$

where n represents an integer denotation. The layout of A in memory is:



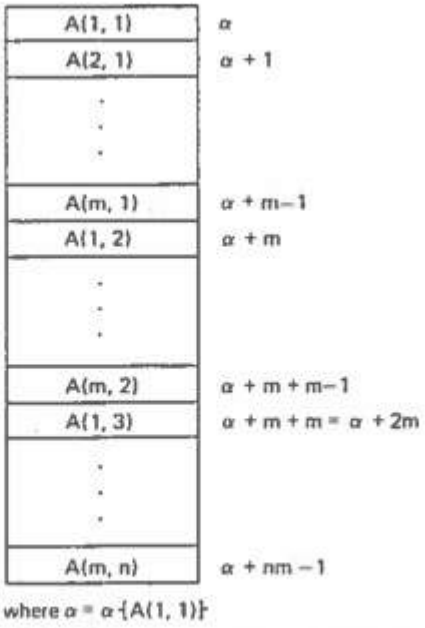


Figure 2.7 Column-Major Layout of Two-Dimensional Array

FORTRAN PROGRAM

```
PROGRAM INOUT
C
C This program reads in and prints out a name
C
CHARACTER NAME*20
PRINT *,' Type in your name, up to 20 characters'
PRINT *,' enclosed in quotes'
READ *,NAME
PRINT *,NAME
END
```

PROGRAM AVERAGE

```
C
C THIS PROGRAM READS IN THREE NUMBERS AND SUMS AND
AVERAGES THEM.
C
REAL NUMBR1,NUMBR2,NUMBR3,AVRAGE,TOTAL
INTEGER N
N = 3
TOTAL = 0.0
PRINT *,'TYPE IN THREE NUMBERS'
PRINT *,'SEPARATED BY SPACES OR COMMAS'
READ *,NUMBR1,NUMBR2,NUMBR3
TOTAL= NUMBR1+NUMBR2+NUMBR3
AVRAGE=TOTAL/N
PRINT *,'TOTAL OF NUMBERS IS',TOTAL
PRINT *,'AVERAGE OF THE NUMBERS IS',AVRAGE
END
```

PROGRAM CUBE ROOT OF FIRST 10 NATURAL NUMBERS

```
C
C THIS PROGRAM FINDS THE CUBEROOT OF FIRST 10 NATURAL NUMBERS
C

DO 100 I = 1, 10
   PRINT *, i**1.0/3
   100 CONTINUE
 END DO
100 CONTINUE
END
```

SUM OF ALL EVEN NUMBER

```
   REAL OUTPUT
   OUTPUT=0.0
   DO I = 2, 100, 2

   OUTPUT = OUTPUT + I
   CONTINUE
   END DO
   PRINT *, OUTPUT
```

FORTRAN IV breaks the machine dependent features but on doing so it violates the portability principle.

FORTRAN was originally designed as programming language for the IBM 704 computers; it was thought that there would be similar, but different languages for other computers. But FORTRAN was used not only for IBM 704 but also for the computers of other manufacturers.

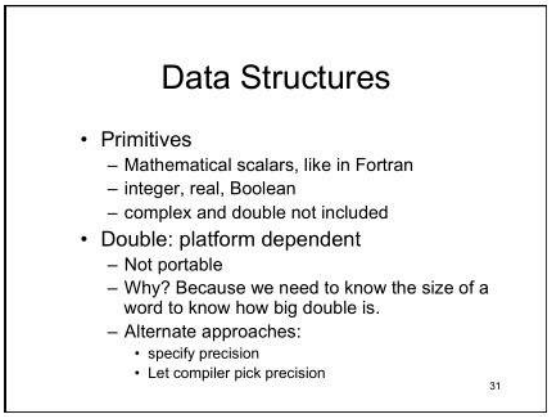
Since FORTRAN was specially designed for the IBM 704, it had many similarity with IBM 704 instruction set. But FORTRAN IV breaks such machine dependent features with more unusual control structures like IF-statement.

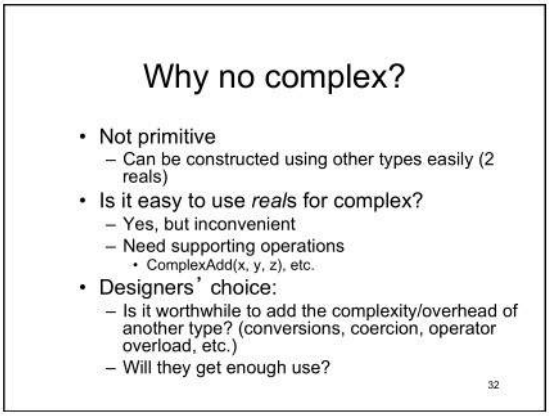
IF (e) n1, n2, n3,
It evaluates expression e and branch to n1, n2, n3 depending on whether the result of the e is -ve, 0 or +ve. This is exactly the function of 704's CAS instruction.

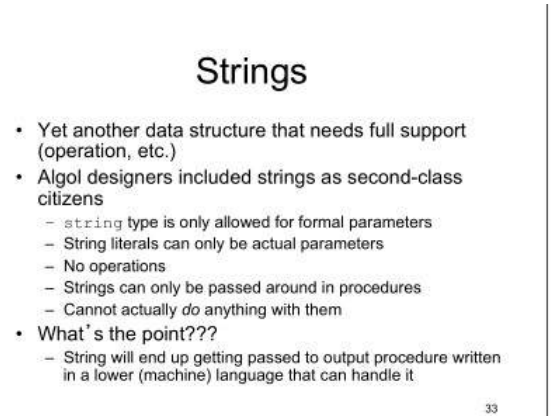
Solved in later version of FORTRAN
IF (X .EQ. A(I)) K = I – 1

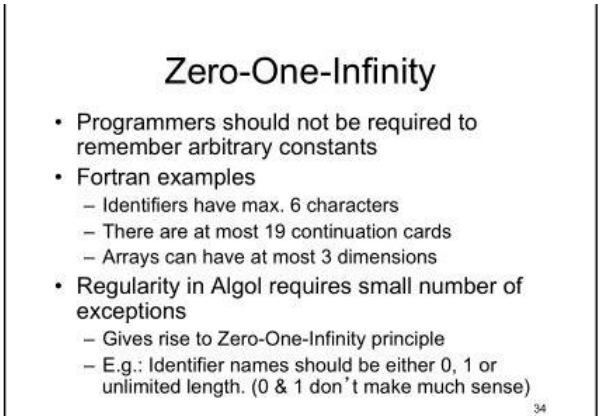
Data Structures in ALGOL-60

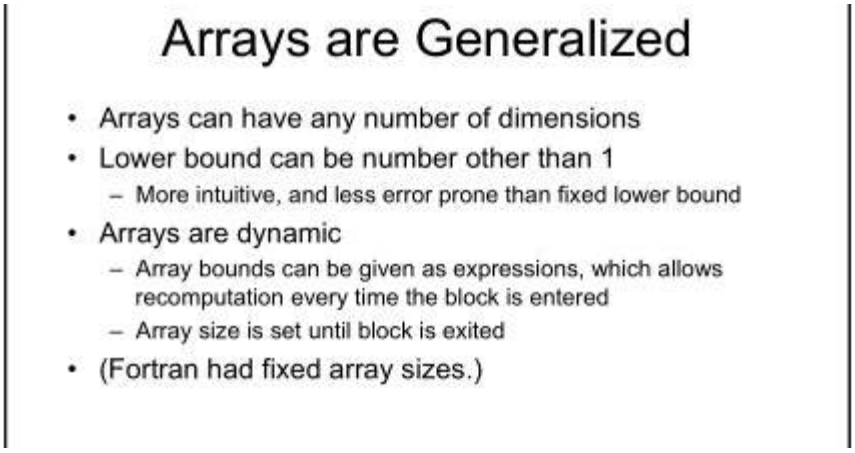
Like FORTRAN, Algol**-60** intended to be used in scientific applications. Therefore the primitive data types are **integer, real and Boolean.**











Integer array NumberOfDays [-100, 200]

Parameter Passing in Algol-60

Parameter Passing

- Modes in Algol
  - Pass by value
  - Pass by name
- Two modes attempt to distinguish between *input only* and *input/output* parameters

51

Pass by Value

```
integer procedure fac(n);
  value n; integer n;
```

- First part of pass by value-result (in Fortran)
  - Actual copied into variable corresponding to formal
  - Secure; local variable will not overwrite actual parameter
  - Does not allow output parameters (input only)
  - Inefficient for arrays (or other non-primitive data structures, in general)
    - Copy must be made of entire array in activation record
    - Copying takes time

52

Pass by Name

- Based on substitution
  - Consider

```
integer procedure Inc(n);
  integer n;
  n := n + 1;
```

  - And the call `Inc(i)`
- We need output parameter that will effect `i`, not just local `n`
  - Acts like `i` is substituted for `n`

```
i := i + 1
```

53

Copy Rule

- Procedure can be replaced by its body with actuals substituted for formals
- Revised Report 4.7.3
- Body of `Inc(n)`
  - `i := i + 1`
  - `A[k] := A[k] + 1`
- Not how it is implemented

54

Pass by Name is Powerful

- Evaluate the following using pass by value, reference, and name

```
procedure S(el,k);
  integer el, k;
  begin
        k := 2;
        el := 0;
  end
A[1] := A[2] := 1;
i := 1;
S(A[i], i)
```

- Value A[1] = 1, A[2] = 1, i = 1
- Reference A[1] = 0, A[2] = 1, i = 2
- Name A[1] = 1, A[2] = 0, i = 2

55

"Thunks"

- Implementing pass by name
  - Passing the text?
    - Would need to compile at runtime
      - not possible
  - Copying compiled code?
    - Would increase size of code…
  - Solution: "Thunks"
    - Pass address to compiled code
    - Address of memory location is returned to callee to use as variable

56

Pass by Name is Dangerous!

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;
        t := x;
        x := y;
        y := t;
  end
```

- What is the effect of
  - Swap(A[i], i)?
  - Swap(i, A[i])?

- Swap(r,s), where r=1,s=2

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;
        t := r;     t=1
        r := s;     r=2
        s := t;     s=1
  end
```

- Swap(A[i], i) where A[i]=27, i=1

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;
        t := A[i];  t=27
        A[i] := i;  A[1]=1
        i := t;     i=27
  end
```

- Swap(i, A[i]), where i=1, A[i]=27

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;
        t := i;     t=1
        i := A[i];  i=27
        A[i] := t;  A[27]=1
  end
```

Name Structure in ALGOL-60

As data structure organize the Data, control structure organizes the control flow of program, name structure organizes the names that appear in a program.

When we declare a new variable in a program:

INTEGER I

It bound the identifier I to a location in a memory. Here INTEGER specify the type of the variable, and I is the identifier.

Declarative constructs are the primitives name structure of ALGOL-60.

The array declaration of algol-60 is similar to that of FORTRAN but it allowed the index to be other than 1.

real array Data[-50, 50]

- Algol-60 introduces the compound statement
  - Where 1 statement is allowed, more can be used, using begin-end

```
for i := 1 step 1 until N do
      sum := sum + Data[i]

for i := 1 step 1 until N do
 begin
      sum := sum + Data[i];
      Print Real (sum)
 end
```

  - Also, the body of a procedure is a single statement

Algol was a major milestone in programming language
Thought it directly competed with FORTRAN and was never widely used, it is one of the major milestone in programming language development.

Terms it added in programming

➔ Type, formal parameter, actual parameter, block, call-by-value, call-by-name, scope, dynamic array, global and local variables.

Concepts it added in programming

➔ Activation Records, Thunks, displays, static and dynamic chains

Influence on succedding programming language
Adaptation of BNF in mathematical theory.

Algol changed the way of programming in effiecient way
Terms it added in programming
Concepts it added in programming
Machine independence lead to a free format
Stack allocation permits dynamic array
Nesting led to structured programming.

BNF and EBNF

The use of natural language like English in defining the syntactic notation is very inadequate.

For eg to define a number in English, first we have to define digits, unsigned integer, integer, decimal fraction, exponent part, decimal number, unsigned number. Then only we can define number as unsigned number followed by +, unsigned number followed by – or unsigned number followed by no sign.

Unless we have some examples, the above definition is useless.
Digit: '0', '1', … '9'
Unsigned integer: '273', '33'
Integer: '-25', '38'
Decimal fractions: '.02', '.3'
Exponent form: '10^23'
Decimal number: '273', '3.1'
Unsigned number: '273', '3.0'
Number: '-2', '3.0'

But this is also not engouh as there will always be a case when the examples are not enough.

Backus-Naur-Form is Naur's adaptation of the Backus notation to the Algol-60 report. In BNF we combine the power of using the Natural language to define the syntactic notation and examples to clarify things. BNF is a descriptive metalanguage because it describe another langauge, the object language.

To define a decimal fraction:
<decimal fraction> ::= .<unsigned integer>

::= symbol refers to “is defined as”

Use of “|” for alternation
<integer> ::= +<unsigned integer>
| -<unsigned integer>
| <unsigned integer>

Use of recursion to denote the sequence of one or more things
<unsigned integer> ::= <digit>
| <unsigned integer><digit>

```
<unsigned integer>  ::=  <digit>
                       |  <unsigned integer>  <digit>

<integer>    ::=  +  <unsigned integer>
                |  −  <unsigned integer>
                |     <unsigned integer>

<decimal fraction>  ::=  .  <unsigned integer>

<exponent part>  ::=  10  <integer>

<decimal number>  ::=  <unsigned integer>
                     |  <decimal fraction>
                     |  <unsigned integer>  <decimal fraction>

<unsigned number>  ::=  <decimal number>
                      |  <exponent part>
                      |  <decimal number>  <exponent part>

<number>    ::=  +  <unsigned number>
                |  −  <unsigned number>
                |     <unsigned number>
```

Figure 4.1 BNF Definition of Numeric Denotations

Extended BNF Is More Descriptive

Several extensions have been made to BNF to improve its readability. Often these directly reflect the words and phrases commonly used to describe syntax. For example, in the prose description of *number*, we defined an *unsigned integer* to be a "sequence of one or more *digits*." This was expressed in BNF through a recursive definition. Although we can get used to recursive descriptions such as this, it would really be more convenient to have a notation that directly expresses the idea "sequence of one or more of . . . ." One such notation is the *Kleene cross*,[2] C^+, which means a sequence of one or more strings from the syntactic category C. Using this notation, the syntactic category <unsigned integer> can be defined:

<unsigned integer> ::= <digit>$^+$

A useful variant of this notation is the *Kleene star*, C^*, which stands for a sequence of *zero* or more elements of the class C. For example, an Algol identifier (name) is a <letter> followed by any number of <letter>s or <digit>s (i.e., a sequence of zero or more <alphanumeric>s). This can be written as:

<alphanumeric> ::= <letter> | <digit>

<identifier> ::= <letter> <alphanumeric>*

If <alphanumeric> is used in only one place, namely, the definition of <identifier>, then it is pointless to give it a name. Rather, it would be better to be able to say directly a "sequence of <letter>s or <digit>s." Some dialects of BNF permit this by allowing us to substitute '{<letter> | <digit>}' for <alphanumeric>:

<identifier> ::= <letter> {<letter> | <digit>}*

This can be read, "An identifier is a letter followed by a sequence of zero or more letters or digits." It can be made even more pictorial by stacking the alternatives:

$$\text{<identifier>} ::= \text{<letter>} \left\{ \begin{array}{c} \text{<letter>} \\ \text{<digit>} \end{array} \right\}^*$$

2 Pronounced "klane-uh" and named after the mathematician and logician S. C. Kleene (1909–).

Let's see if there are any other improvements that can be made to this notation. The rule for an <integer> is

```
<integer>  ::= + <unsigned integer>
             | - <unsigned integer>
             |   <unsigned integer>
```

One immediate simplification should be apparent; we can use our alternative notation

$$\langle \text{integer} \rangle ::= \left\{ \begin{array}{c} + \\ - \end{array} \right\} \langle \text{unsigned integer} \rangle$$
$$\quad | \quad \langle \text{unsigned integer} \rangle$$

This is better; but the idea that we really want to express is that the <unsigned integer> is *optionally* preceded by a '+' or '−'. The square bracket is often used for this purpose, that is, '[+ | −]' or

$$\langle \text{integer} \rangle ::= \left[\begin{array}{c} + \\ - \end{array} \right] \langle \text{unsigned integer} \rangle$$

This is a very graphic notation; it *shows* us what an integer looks like. A two-dimensional metalanguage like this was first used in the 1960s to describe the COBOL language. Figure 4.2 shows the definition of <number> using this extended BNF.

Why is this extended BNF preferable to pure BNF? It is because it adheres better to the Structure Principle, that is, the forms of the extended BNF defini-

$$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle^{+}$$

$$\langle \text{integer} \rangle ::= \left[\begin{array}{c} + \\ - \end{array} \right] \langle \text{unsigned integer} \rangle$$

$$\langle \text{decimal number} \rangle ::= \left\{ \begin{array}{l} \langle \text{unsigned integer} \rangle \\ [\langle \text{unsigned integer} \rangle] \,.\, \langle \text{unsigned integer} \rangle \end{array} \right\}$$

$$\langle \text{number} \rangle ::= \left[\begin{array}{c} + \\ - \end{array} \right] \left\{ \begin{array}{l} \langle \text{decimal number} \rangle \\ [\langle \text{decimal number} \rangle]_{10} \ \langle \text{integer} \rangle \end{array} \right\}$$

Figure 4.2 Extended BNF Definition of Numeric Denotations

Context free and regular grammars

There are mathematical definitions of context-free and regular languages, but the simplest way to see the difference is by reference to the extended BNF notation. A *regular grammar* is one that is written in extended BNF without the use of any recursive rules. For example, the definition of <number> in Figure 4.2 is an example of a regular grammar. A language is formally defined as the set of strings described by a grammar. Therefore, a *regular language* is a language that can be described by a regular grammar. For example, the language defined by <number>, which is {'−273', '$6.02_{10}23$', '3.14159', '$+3_{10}8$',...} is a regular language. Notice that we said that a regular language is one that *can* be described by a regular grammar, not one that *must* be. The distinction is important since the grammar in Figure 4.1 uses recursion and defines the same language as that in Figure 4.2.

A *context-free* grammar is any grammar that can be expressed in extended BNF, possibly including recursively defined nonterminals. Analogously, a *context-free language* is a language that can be described by a context-free grammar. All regular grammars are also context-free grammars, and all regular languages are also context-free languages.

Thus, the difference between regular and context-free grammars is in the use of recursion in the definitions. What are the implications of this? Basically, recursion allows the definition of *nested* syntactic structures. We have already seen the importance of nesting in Algol. Nesting is easy to express in BNF, which probably encouraged the widespread use of nested constructs in Algol. Consider this simplified form of Algol's definition of a <statement>:

<unconditional statement> ::=

{ <assignment statement>
for <for list> **do** <statement> }

<statement> ::=

{ <unconditional statement>
if <expression> **then** <unconditional statement>
if <expression> **then** <unconditional statement> **else** <statement> }

We can see that <statement> is defined recursively in terms of itself since part of an **if**-statement is itself a <statement>. Also, <statement> is defined in terms of <unconditional statement>, which is in turn defined in terms of <statement>; this is an example of *indirect recursion*. This recursive definition is not circular because the <assignment statement> alternative (which is not defined in terms of <statement>) provides a *base* for the recursion.

It is easy to see that these definitions allow the nesting of statements. For example, since 'x = x + 1' is an <assignment statement>, it is also an

<unconditional statement>. Therefore, it can be made part of an **if**-statement, for instance,

```
if x<0 then x := x+1
```

Since this **if**-statement is itself a <statement>, it can be made the part of a **for**-loop, for example,

```
for i := 1 step 1 until n do if x<0 then x := x+1
```

This process of embedding statements into yet larger statements can be continued indefinitely. Before the development of the BNF notation, it would have been much more complicated to describe these recursively nested structures.

Since regular grammars do not permit recursion, it is not possible to specify indefinitely deep nesting in a regular grammar. Thus, the major difference between regular and context-free languages is the possibility of indefinitely deep nesting in the latter. A simple example of this is the language of balanced parentheses. This is a make-believe language whose only strings are sequences of properly balanced parentheses. For example,

(((()))(()((()())))

is an expression in this language, while '((((()' is not. This language is easy to describe as a context-free grammar:

<expression> ::= <balanced>*

<balanced> ::= (<expression>)

(*N.B.* It may look as though there is no base to the recursive definitions above, but that is not the case since the Kleene star notation allows *no* repetitions of <balanced> to be an <expression>.)

Algol Solved Some Problems of FORTRAN Lexics

Algol's distinction among reference, publication, and hardware languages led to the idea that the keywords of the language are indivisible basic symbols. That is, boldface (or underlined) words such as 'if' and '**procedure**' are considered single symbols, like '$\leqslant$' and ':=', and have no relation to similarly appearing identifiers such as 'if' or 'procedure'. This means that the confusion between keywords and identifiers that we saw in FORTRAN is not possible in Algol. For example, it is perfectly unambiguous (although somewhat confusing) to write:

if procedure **then** until := until + 1 **else** do := **false**;

There are many possible hardware representations for these compound symbols, including,

'if' procedure 'then' until := until + 1 'else' do := 'false';

IF procedure THEN until := until + 1 ELSE do := FALSE;

#IF PROCEDURE #THEN UNTIL := UNTIL + 1 #ELSE DO := #FALSE;

if Xprocedure then Xuntil := Xuntil + 1 else Xdo := false;

The last example illustrates the *reserved word* approach; the symbols used for keywords are not allowed as identifiers. Although this seems to be the most convenient convention, it really violates the spirit of Algol since the keywords can conflict with the identifiers.

It is useful to distinguish the following three lexical conventions for the words of a programming language:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

must be written

```
(quotient    (plus    (minus b)
                      (sqrt (difference    (expt b 2)
                                           (times 4 a c))))
             (times 2 a))
```

Common LISP (and many other dialects) permits symbolic operations, but it's not much of an improvement:

```
(/ (+ (- b) (sqrt (- (expt b 2) (* 4 a c))))
   (* 2 a))
```

On the other hand, it is fairly easy to write a LISP function to translate conventional infix notation into LISP's prefix notation. If we did this, we could write (assuming the function was named 'infix' and special characters are allowed as atoms):

```
(infix '( ( - b + sqrt ( b ↑ 2 - 4 * a * c )) / (2 * a) ))
```

Car and Cdr acces the part of the list

The first element of a list is selected by the 'car' function.' For example,

```
(car '(to be or not to be) )
```

returns the atom 'to'. The first element of a list can be either an atom or a list, and 'car' returns it, whichever it is. For example, since Freq is the list

```
( (to 2) (be 2) (or 1) (not 1) )
```

the application

```
(car Freq)
```

returns the list

```
(to 2)
```

Notice that the argument to 'car' is always a nonnull list (otherwise it can't have a first element) and that 'car' may return either an atom or a list, depending on what its argument's first element is.

Since there are only two selector functions and since a list can have any number of elements, any of which we might want to select, it's clear that 'cdr' must provide access to the rest of the elements of the list (after the first).

The 'cdr'[6] function returns all of a list *except* its first element. Therefore,

```
(cdr '(to be or not to be) )
```

returns the list

```
(be or not to be)
```

Similarly, '(cdr Freq)' returns

```
( (be 2) (or 1) (not 1) )
```

'Car' and 'cdr' can be used in combination to access the components of a list. Suppose DS is a list representing a personnel record for Don Smith:

```
(set 'DS '( (Don Smith) 45 30000 (August 25 1980) ) )
```

The list DS contains Don Smith's name, age, salary, and hire date. To extract the first component of this list, his name, we can write '(car DS)', which returns '(Don Smith)'. How can we access Don Smith's age? Notice that the 'cdr' operation deletes the first element of the list, so that the second element of the original list is the first element of the result of 'cdr'. That is, '(cdr DS)' returns

```
(45 30000 (August 25 1980))
```

so that '(car (cdr DS))' is 45, Don Smith's age. We can now see the general pattern: To access an element of the list, use 'cdr' to delete all of the preceding elements and then use 'car' to pick out the desired element. Therefore, '(car (cdr (cdr DS)))' is Don Smith's salary, and

```
(car (cdr (cdr (cdr DS))))
```

is his hire date. We can see this from the following (by 'cdr ⇒' we mean "applying 'cdr' returns"):

```
( (Don Smith) 45 30000 (August 25 1980) )
cdr ⇒ (45 30000 (August 25 1980) )
```

In general, the nth element of the list can be accesed by n-1 cdrs followed by car.

We can see that any part of a list structure, no matter how complicated, can be extracted by appropriate combinations of 'car' and 'cdr'. This is part of the simplicity of LISP; just these two selector functions are adequate for accessing the components of any list structure. This can, of course, lead to some large compositions of 'car's and 'cdr's, so LISP provides an abbreviation. For example, an expression such as

```
(car (cdr (cdr (cdr DS))))
```

can be abbreviated

```
(cadddr DS)
```

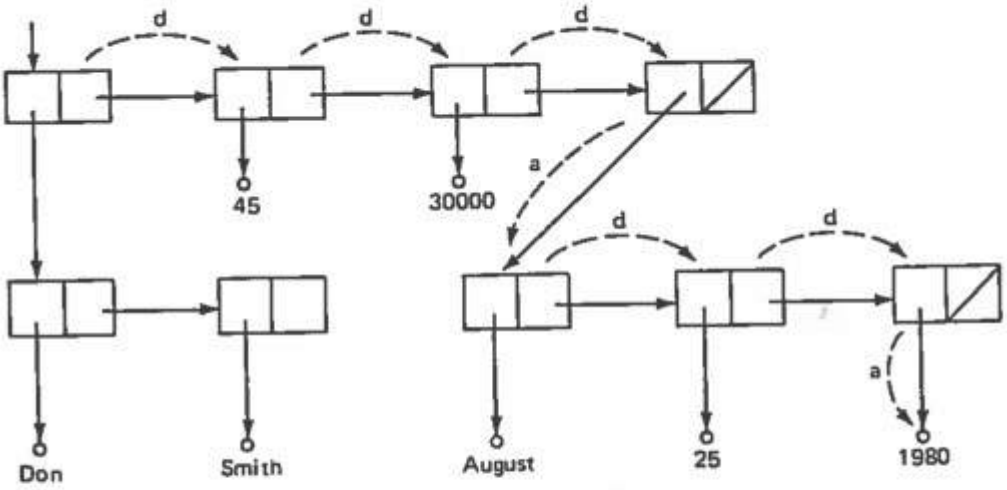
The composition of 'car's and 'cdr's is represented by the sequence of 'a's and 'd's between the initial 'c' and the final 'r'. By reading the sequence of 'a's and 'd's in reverse order, we can use them to "walk" through the data structure. For example, 'caddr' accesses the salary:

```
((Don Smith) 45 30000 (August 25 1980))
d ⇒ (45 30000 (August 25 1980))
d ⇒ (30000 (August 25 1980))
a ⇒ 30000
```

Also, 'cadar' accesses the last-name component of the list:

```
((Don Smith) 45 30000 (August 25 1980))
```

This can be seen more clearly if the list is written as a linked data structure; then a 'd' moves to the right and an 'a' moves down:



This shows that '(caddadddr DS)' accesses the year Don Smith was hired. Clearly, these sequences of 'a's and 'd's can become quite complicated to read. Writing them is also error-prone. One solution to this is to write a set of special-purpose functions for accessing the parts of a record. For example, a function for accessing the hire date could be defined as:

```
(defun hire-date (r) (caddddr r))
```

Property Lists

Property List is the method of representing information where property indicator is followed by property value. Its format is

(p1 v1 p2 v2 p3 v3 …. pn vn)

In which pi is the property indicator and vi is the property value.

For example:

(name (Don Smith) age 45 salary 30000 hire-date (August 25 1980))

The advantage of property lists is their flexibility; as long as the properties are accessed by their indicators, programs will be independent of the particular arrangement of the data. For eg, p-list below is same as above:

(age 45 name (Don Smith) salary 30000 hire-date (August 25 1980))

Find the property value using the property indicator

Algorithm:

```
(getprop p x) =
  if (eq (car x) p)
    then return (cadr x)
    else return (getprop p (cddr x))
```

It only remains to translate this into LISP notation. A LISP conditional expression is written:

(cond (c_1 e_1) (c_2 e_2) ··· (c_n e_n))

The conditions c_1, c_2, …., c_n are evaluated in order until one returns 't'. The value of the conditional is the value of the corresponding e_i. The effect of an "else" clause is accomplished by using 't' for the last condition. The 'getprop' function is:

```
(defun getprop (p x)
  (cond ((eq (car x) p) (cadr x))
        (t (getprop p (cddr x))) ))
```

To find out properties of

Information Can Be Represented in Association Lists

The property list data structure described on pp. 349–351 works best when exactly one value is to be associated with each property. That is, a property list has the form:

$$(p_1\ v_1\ p_2\ v_2\ \cdots\ p_n\ v_n)$$

This is sometimes inconvenient; for example, some properties are *flags* that have no associated value—their presence or absence on the property lists conveys all of the information. In our personnel record example, this might be the 'retired' flag, whose membership in the property list indicates that the employee has retired. Since property indicators and values must alternate in property lists, it is necessary to associate *some* value with the 'retired' indicator, even though it has no meaning.

An analogous problem arises if a property has several associated values. For example, the 'manages' property might be associated with the names of everyone managed by Don Smith. Because of the required alternation of indicators and values in property lists, it will be necessary to group these names together into a subsidiary list.

These problems are solved by another common LISP data structure—the *association list*, or *a-list*. Just as we can associate two pieces of information in our minds, an association list allows information in list structures to be associated. An *a*-list is a list of pairs,' with each pair associating two pieces of information. The *a*-list representation of the properties of Don Smith is:

```
( (name (Don Smith))
  (age 45)
  (salary 30000)
  (hire-date (August 25 1980)) )
```

The general form of an *a*-list is a list of attribute-value pairs:

$$((a_1\ v_1)\ (a_2\ v_2)\ \ldots\ (a_n\ v_n))$$

As for property lists, the ordering of information in an *a*-list is immaterial. Information is accessed *associatively*; that is, given the indicator 'hire-date', the associated information '(August 25 1980)' can be found. It is also quite easy to go in the other direction: Given the "answer" '(August 25 1980)', find the "question," that is, 'hire-date'. The function that does the forward association is normally called 'assoc'. For example,

```
(set 'DS '((name (Don Smith) (age 45) ...) )
  ((name (Don Smith)) (age 45) ...)
(assoc 'hire-date DS)
  (August 25 1980)
(assoc 'salary DS)
  30000
```

Assoc

```
(defun assoc (at l)
   (cond
     ((null l) nil)
     ((eq at (caar l)) (car l))
     (t (assoc at (cdr l)))))
```

Data Structure in Lisp

1.1 Data Structures

LISP data structures are called "S-expressions." The S stands for "symbolic." In this text, the terms "S-expression" and "expression" are used interchangeably. An S-expression is

1. a *number*, e.g., `15`, written as an optional plus or minus sign, followed by one or more digits.
2. a *symbol*, e.g., `FOO`, written as a letter followed by zero or more letters or digits.

3. a *string*, e.g., `"This is a string"`, written as a double quote, followed by zero or more characters, followed by another double quote.
4. a *character*, e.g., `#\q`, written as a sharp sign, followed by a backslash, followed by a character. (Numbers, symbols, strings, and characters are called *atoms*. There are other kinds of atoms. We will see them in later chapters.)
5. a *list* of S-expressions, e.g., `(A B)` or `(IS TALL (FATHER BILL))`, written as a left parenthesis, followed by zero or more S-expressions, followed by a right parenthesis.

Parentheses are more significant in LISP than they are in most other programming languages. Parentheses are virtually the only punctuation marks available in LISP programs. They are used to indicate the structure of S-expressions. For example, `(A)` is a list of one element, the symbol `A`. `((A))` is also a list of one element, which is in turn a list of one element, which is the symbol `A`. Notice also that the left and right parentheses must *balance*. That is, a well-formed S-expression has a right parenthesis to close off each left parenthesis.

Structural Organization of Lisp

11.1 Introduction to Symbol Expressions

The S-expression

The syntactic elements of the Lisp programming language are *symbolic expressions*, also known as *s-expressions*. Both programs and data are represented as s-expressions: an s-expression may be either an *atom* or a *list*. Lisp atoms are the basic syntactic units of the language and include both numbers and symbols. Symbolic atoms are composed of letters, numbers, and the non-alphanumeric characters.

```
(defun make-table (text table)
  (cond    ((null text) table)
           (t (make-table  (cdr text)
                           (update-entry table (car text)) )) ))

(defun update-entry (table word)
  (cond    (( null table) (list (list word 1)) )
           ((eq word (caar table))
           (cons (list word (add1 (cadar table)))
                 (cdr table)))
           (t (cons (car table)
                    (update-entry (cdr table) word))) ))

(defun lookup (table word)
  (cond    ((null table) 0)
           ((eq word (caar table)) (cadar table))
           (t (lookup (cdr table) word)) ))

(set 'text '(to be or not to be))

(set 'Freq (make-table text nil))
```

Figure 9.1 Example of LISP Program

Function application is the central idea.

Plus 2 3

List is the primary data structure

(set ‘tex ‘((to be or not to be)))

Polish Notation and hierarchial structure in Lisp

To see this, it's necessary to know that a function application in LISP is written as follows:

$$(f\ a_1\ a_2\ \ldots\ a_n)$$

where f is the function and $a_1, a_2, \ldots, a_n$ are the arguments. This notation is called *Cambridge Polish* because it is a particular variety of Polish notation developed at MIT (in Cambridge, Mass.). *Polish notation* is named after the Polish logician Jan Lukasiewicz.

The distinctive characteristic of Polish notation is that it writes an operator before its operands. This is also sometimes called *prefix notation* because the operation is written before the operands (pre = before). For example, to compute 2 + 3 we would type[2]

```
(plus 2 3)
```

to an interactive LISP system, and it would respond

```
5
```

LISP's notation is a little more flexible than the usual infix notation since one 'plus' can sum more than two numbers. We can write

```
(plus 10 8 5 64)
```

2.2.2 Hierarchical Structures

The representation of sequences in terms of lists generalizes naturally to represent sequences whose elements may themselves be sequences. For example, we can regard the object `((1 2) 3 4)` constructed by

```
(cons (list 1 2) (list 3 4))
```

```
((1 2) 3 4)
```

as a list of three items, the first of which is itself a list, `(1 2)`. Indeed, this is suggested by the form in which the result is printed by the interpreter. Figure 2-5 shows the representation of this structure in terms of pairs.

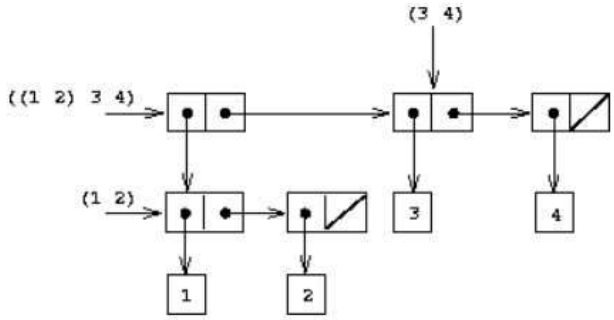


Figure 2.5: Structure formed by `(cons (list 1 2) (list 3 4))`.

Another way to think of sequences whose elements are sequences is as `trees` . The elements of the sequence are the branches of the tree, and elements that are themselves sequences are subtrees. Figure 2-6 shows the structure in Figure 2-5 viewed as a tree.

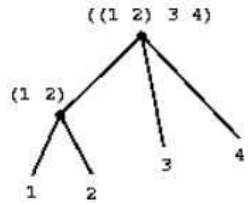


Figure 2.6: The list structure in Figure 2-5 viewed as a tree.

The Conditional Expression Is a Major Contribution

In the historical discussion of LISP, we mentioned that LISP was the first language to contain a conditional expression. This was an important idea since it meant that everything could be written as an expression. Previous languages, such as FORTRAN, and some newer languages, such as Pascal, require the user to drop from the expression level to the statement level in order to make a choice. Languages that force expressions to be broken up in this way can often make programs less readable. Mathematicians have recognized for many years the value of conditional expressions and have often used them. For example, here is a typical definition of the *signum* (sign) function:

$$sg(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$$

In LISP this function is defined:

```
(defun sg (x)
  (cond ((plusp x)      1)
        ((zerop x)      0)
        ((minusp x)     -1)) )
```

The LISP conditional has more parentheses than are really necessary; instead of

(cond $(p_1\ e_1) \ldots (p_n\ e_n)$)

it would have been quite adequate if LISP had been designed to use

(cond $p_1\ e_1 \ldots p_n\ e_n$)

However, the latter notation complicates the interpreter slightly and, as we said in the beginning of Chapter 9, nobody imagined that the S-expression notation would become the standard way of writing LISP programs.

The Logical Connectives Are Evaluated Conditionally

McCarthy took the conditional expression as one of the fundamental primitives of LISP. Therefore, it was natural to define the other logical operations in terms of it. For example, the function '(or x y)' has the value 't' (true) if either or both of x and y have the value 't'; in any other case, 'or' has the value 'nil' (false). Another

way to say this is if *x* has the value 't', then the 'or' has the value 't', otherwise the 'or' has the same value as *y*. Write out a truth table for each of these definitions of 'or' to see that they're equivalent. The latter definition allows 'or' to be defined as a conditional:

```
(or x y) = (cond (x t) (t y))
```

Mapcar

Thus, we will define a function 'mapcar' that applies a given function to each element of a list and returns a list of the results. It is not hard to see how to do this since the pattern of the recursion is just an abstraction from the patterns of 'add1-map' and 'zerop-map':

```
(defun mapcar (f x)
  (cond  ((null x) nil)
         (t (cons (f (car x)) (mapcar f (cdr x)) )) ))
```

The function is called 'mapcar' because it applies 'f' to the 'car' of the list each time. Most LISP systems provide 'mapcar' for the user, although on some the order of the arguments is reversed. There's not much standardization in the LISP world!

With this definition of 'mapcar' our previous examples can be expressed without having to write a recursive definition:

```
(mapcar 'add1 '(1 9 8 4))
  (2 10 9 5)
(mapcar 'zerop '(4 7 0 3 -2 0 1))
  (nil nil t nil nil t nil)
(mapcar 'not (mapcar 'zerop '(4 7 0 3 -2 0 1)))
  (t t nil t t nil t)
```

Reduce

```
(defun reduce  (f  a  x)
(cond
        (null x) a
        (t (f  (car  x)a) (reduce  f  a  (cdr  x) )) ) )
```

(reduce 'plus 0' (1 2 3 4 5))

15

Object Representation in Smalltalk

Everything in Smalltalk is an object. In Smalltalk even the classes are objects, i.e they are the instance of the class named class. Objects are the things that represents the quantities, properties and the entities modeled by the program.

Representation of object follows the information hiding and abstraction principle. The representation of object must contain just that information that varies from object to object; the similar information from object to object are stored in the definition of the class. We also are not able to access the information of an object unless we know what the class of the object is.

Let's see the example shown below, which shows the representation of two boxes B1 and B2. To keep the figure clear, we have abbreviated or omitted some of the component of objects. For example, the representation of object 500@200 is shown but the representation of object 5000@600 is abbreviated.

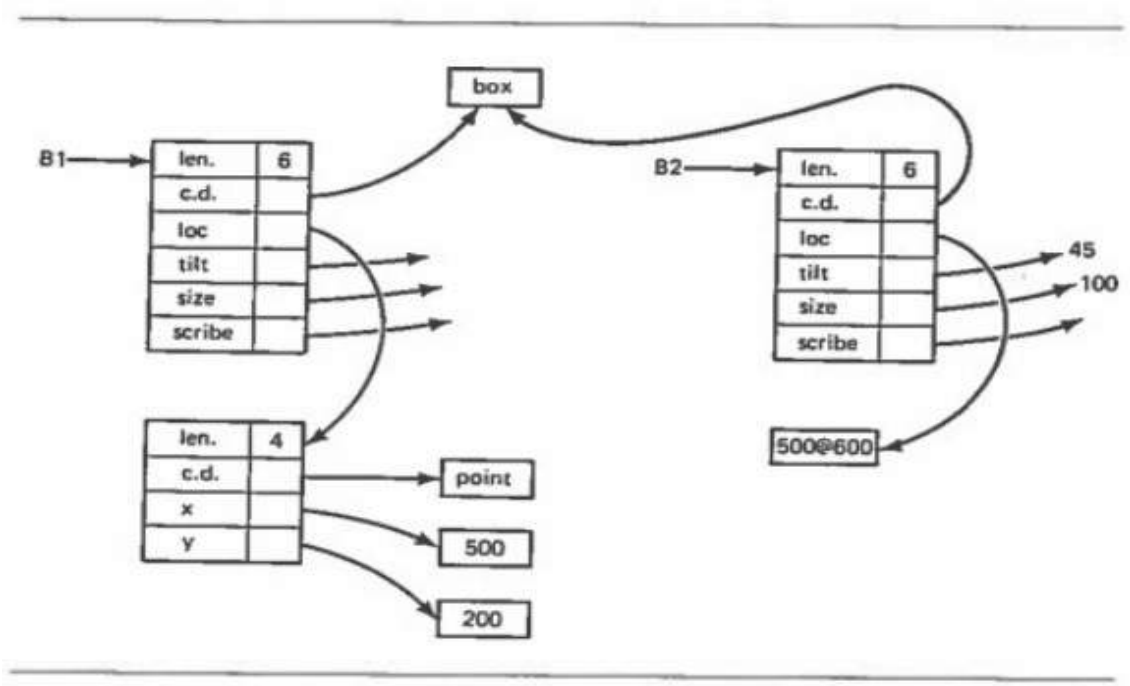


Figure 12.9 Representation of Objects

Class Representation

We have said that everything in Smalltalk is an object. This rule is true without exception. In particular it includes classes, which are just instances of the class named 'class'. Therefore, classes are represented like the objects just described, with length and class description fields (the latter pointing to the class 'class').

The instance variables of a class object contain pointers to objects representing the information that is the same for all instances of the class.

What is this information? We can get a clear idea by looking at a class definition, such as the one in Figure 12.5. The information includes the following:

1. The class name
2. The superclass (which is 'object' if no other is specified)
3. The instance variable names
4. The class message dictionary
5. The instance message dictionary

(Just as there are instance methods and class methods, Smalltalk allows both *instance variables* and *class variables*. We will ignore the latter, beyond mentioning that they are stored in the class object.)

Thus, observations of class definitions lead to a representation like that shown in Figure 12.10. (We have written the class of an object above the rectangle representing that object.)

Notice that we have added a field 'inst. size' (instance size), which indicates the number of instance variables. The number of instance variables is needed by the storage manager when it instantiates an object since this number determines the amount of storage required. If we did not have this field, it would be necessary to count the names in the string contained in the 'inst. vars.' field to determine this information. This would slow down object instantiation too much.

This leaves the message dictionaries for our consideration. In Smalltalk, a method is identified by the keywords that appear in a message invoking that

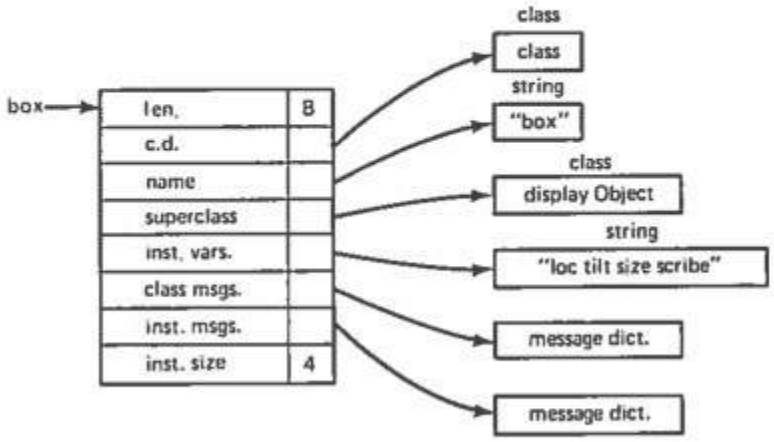


Figure 12.10 Representation of Class Object

Activation Record Representation

There is a strong resemblance between message sending in Smalltalk and procedure calls in other languages. In other languages, activation records are the key vehicle for the procedure calls. Activation record holds the information that pertains to one activation of a procedure, the process of procedure calls and return can be understood as the manipulation of the activation records.

The same is the case in Smalltalk. We will use activation records to hold all of the information relevant to one activation of a method.

Smalltalk activation record have three major parts

- *Environment part*: The context to be used for execution of the method
- *Instruction part*: The instruction to be executed when this method is resumed
- *Sender part*: The activation record of the method that sent the message invoking this method

We will consider these parts in reverse order.

The *sender part* is just the *dynamic link*, that is, a pointer from the receiver's activation record back to the sender's activation record. It is just an object reference since activation records, like everything else in Smalltalk, are objects.

The *instruction part* must designate a particular instruction in a particular method. Since methods are themselves objects (instances of class 'method'), a two-coordinate system is used for identifying instructions:

- An *object pointer* identifies the method-object containing all of the instructions of a method.
- A *relative offset* identifies the particular instruction within the method-object.

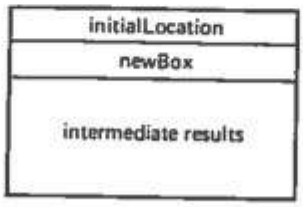
This two-coordinate addressing is necessary because instruction addressing goes through the storage manager (thus adhering to the Information Hiding Principle).

The final part of the activation record is the *environment part*, which must provide access to both the local and nonlocal environments. The local environment includes space for the parameters to the method and the temporary variables. This part of the activation record also must provide space for hidden

temporary variables such as the intermediate results of expressions. For example, the local environment area for the method

```
newAt: initialLocation |newBox|
  newBox ← box new.
    ⋮
```

must contain space for the parameter 'initialLocation', the temporary variable 'newBox', and intermediate results:



The nonlocal environment includes all other visible variables, namely, the instance variables and the class variables. The instance variables are stored in the representation of the object that received the message, therefore, a simple pointer to this object makes them accessible. The object representation contains a pointer to the class of which the object is an instance, therefore, the class variables are also accessible via the object reference. Finally, since the class representation contains a pointer to its superclass, the superclass variables are also accessible. The parts of an activation record are shown in Figure 12.12.

Notice that this approach to accessing nonlocals is very similar to the static chain of environments that we encountered in block-structured languages. There, the static chain led from the innermost active environment to the outermost environment. Here, the static chain leads from the active method to the object that received the message, and from there up through the class hierarchy, to terminate at the class 'object'. Just as in block-structured languages, variable accessing requires knowing the static distance to the variable and skipping that many links of the static chain to get to the environment that defines the variable. (If you are unclear about this, review Chapter 6, Section 6.1.)

EXERCISE 12-13: Show the code for accessing a Smalltalk variable, given its coordinates. Assume the environment coordinate is given as a static distance.

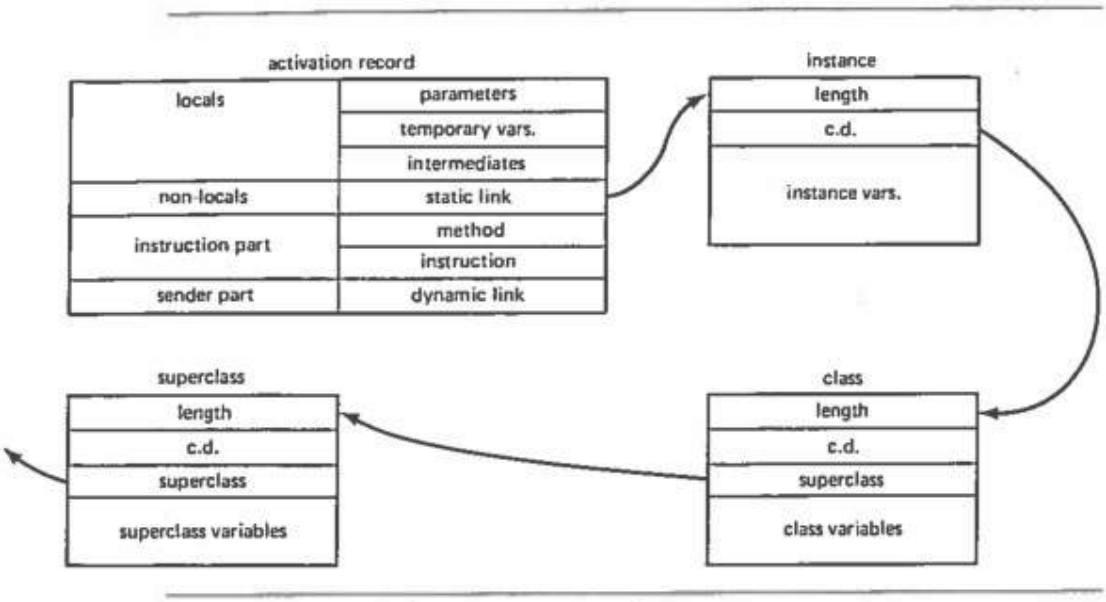


Figure 12.12 Parts of an Activation Record

Message Sending and Returning

We will now list and analyze the steps that must take place when a message is sent to an object:

1. Create an activation record for the receiver (callee).
2. Identify the method being invoked by extracting the template from the message and then looking it up in the message dictionary for the receiving object's class or superclasses. That is, if it's not defined in the class, we must look in the superclass; if it's not defined in the superclass, we must look in its superclass, and so on.
3. Transmit the parameters to the receiver's activation record.
4. Suspend the sender (caller) by saving its state in its activation record.
5. Establish a path (dynamic link) from the receiver back to the sender, and establish the receiver's activation record as the active one.

As we would expect, returning from a method must reverse this process:

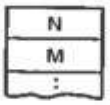
1. Transmit the returned object (if any) from the receiver back to the sender.
2. Resume execution of the sender by restoring its state from its activation record.

We have omitted deallocation of the activation record as part of returning; we explain this next.

In accordance with the Information Hiding Principle, the Storage Manager handles allocation and deallocation of *all* objects; this includes activation record objects. The effect of this is that activation records are created from free storage and reclaimed by reference counting, just like other objects. This is the reason we don't explicitly deallocate activation records. This approach is quite different from the implementation of the other languages we've studied in which activation records occupy contiguous locations on a stack. The Smalltalk approach is a little less efficient, although this is compensated for by its greater simplicity and regularity.

There is another reason that Smalltalk does not allocate its activation records on a stack. The implementation of concurrency we discussed in Section 12.4 has an important limitation: The scheduler must wait for each task to return from its 'run' message before it can schedule the next task. In fact, if one of the tasks went into an infinite loop, the entire system might halt. Therefore, real Smalltalk systems provide an interrupt facility that automatically interrupts the executing task after it has run for a certain amount of time. This ensures that all of the active tasks get serviced regularly. It also means that a runaway task can be deactivated (removed from the 'sched' set) by another task.

What does this have to do with activation records? Consider what must occur when a task is interrupted. Since its execution is being suspended, its state must be stored in its activation record. Now, suppose that Smalltalk's activation records were held on a stack. Further, suppose that in a task *A*, the method *M* is activated; an activation record for *M* will be placed on the top of the stack. Suppose that before *M* returns, task *A* is interrupted and task *B* is resumed. Assume that task *B* activates method *N*, which causes an activation record for *N* to be stacked. The stack now looks like this:



Next, suppose that task *B* is interrupted before *N* returns and that task *A* is resumed. Finally, suppose that task *M* returns and deletes its activation record from the stack. We are faced with two possibilities: Either (1) popping *M* will also pop off *N*'s activation record, which is incorrect since *N* has not returned; or (2) *M* is popped out of the middle of the stack leaving a hole, which means that we're not using a stack after all. The point is that a stack is the appropriate data structure only if we are dealing with something that follows a LIFO (last-in–first-out) discipline. Procedures in a sequential (i.e., nonconcurrent) language follow a LIFO discipline; procedures in a concurrent language do not.

eters are surrounded by parentheses and separated by commas; in Smalltalk parameters are separated by keywords. For example, the Smalltalk message:

```
newBox setLoc: initialLocation tilt: 0 size: 100 scribe: pen new
```

is equivalent to the Ada procedure call:

```
NEWBOX.SET (INITIAL_LOCATION, 0, 100, PEN.NEW() );
```

although the similarity is more striking if we use position-independent parameters:

```
NEWBOX.SET  (LOC => INITIAL_LOCATION, TILT => 0,
             SIZE => 100, SCRIBE => PEN.NEW() );
```

Note, however, that Smalltalk is not following the Labeling Principle here since the parameters are required to be in the right order even though they are labeled.

The message format, keywords followed by colons, can be used if there are one or more parameters to the method. What if a method has no parameters? In this case, it would be confusing to both the human reader and the system if the keyword were followed by a colon. This leads to the format that we have seen for parameterless messages:

```
B1 show
```

Omitting the colon from a parameterless message is analogous to omitting the empty parentheses '( )' from a parameterless procedure call in Ada.

These message formats are adequate for all purposes since they handle any number of parameters from zero on up. Unfortunately, they would require writing arithmetic expressions in an uncommon way. For example, to compute '$(x + 2) \times y$' we would have to write[3]:

```
(x plus: 2) times: y
```

3 The parentheses are necessary, otherwise we would be sending to 'x' a message with the template 'plus:times:'.

To avoid this unusual notation, Smalltalk has made a special exception: the arithmetic operators (and other special symbols) can be followed by exactly one parameter even though there is no colon. For example, in

```
x + 2 * y
```

the object named 'x' is sent the message '+ 2', and the object resulting from this is sent the message '* y'. Thus, this expression computes $(x + 2)y$; notice that Smalltalk does not obey the usual precedence rules.

In summary, there are three formats for messages:

1. Keywords for parameterless messages (e.g., 'B1 show')
2. Operators for one-parameter messages (e.g., 'x + y')
3. Keywords with colons for one- (or more) parameter messages (e.g., 'Scribe grow: 100')

Notice that this format convention fits the Zero-One-Infinity Principle since the only special cases are for zero parameters and one parameter. However, the fact that these cases are handled differently from the general case violates the Regularity Principle. This is a conscious trade-off that the designers of Smalltalk have made so that they can use the usual arithmetic operators. We know this because earlier versions of Smalltalk (e.g., Smalltalk-72) had a uniform method for passing parameters that did not depend on the number of parameters.

Orthogonal Classification

Hierarchical Subclasses Preclude Orthogonal Classifications

We will now discuss one of the limitations of a strictly hierarchical subclass–superclass relationship. Consider an application in which Smalltalk is being used for the computer-aided design of cars. We will assume that this system assists in the design in several different ways. For example, it helps in producing engineering drawings by allowing the user to manipulate and combine diagrams of various parts, and it assists in cost and weight estimates by keeping track of the number, cost, and weight of the parts.

Presumably each part that goes into a car is an object with a number of attributes. For example, a bumper might have a weight, a cost, physical dimensions, the name of a manufacturer, a location on the screen, and an indication of its points of connection with other parts. Similarly, an engine might have weight, cost, dimensions, manufacturer, horsepower, and fuel consumption. If we presume that our display shows only the external appearance of the car, then an engine will not have a display location.

The next step is to apply the Abstraction Principle and to begin to classify the objects on the basis of their common attributes. For example, we will find that many of the classes (e.g., bumpers, roofs, grills) will have a `loc` attribute because they will be displayed on the screen. We can also presume that these objects respond to the `goto:` message so that they can be moved on the screen. This suggests that these classes should be made subclasses of `displayObject` because this class defines the methods for handling displayed objects. Thus, our (partial) class structure might look something like that in Figure 12.8. On the other hand, many of the objects that our program manipulates have `cost`, `weight`, and `manufacturer` attributes. This suggests that we should have a class called, for example, `InventoryItem` that has these attributes and that responds to messages for inventory control (e.g., `reportStock`, `reorder`). This leads to the class structure shown in Figure [illegible].

Now we can see the problem. Smalltalk organizes classes into a hierarchy; each class has exactly one immediate superclass. Notice that in our example several of the classes (`bumper` and `grill`) are subclasses of two classes: `displayObject` and `inventoryItem`. This is not possible in Smalltalk; when a class is defined, it can be specified as an immediate subclass of exactly *one* other class.

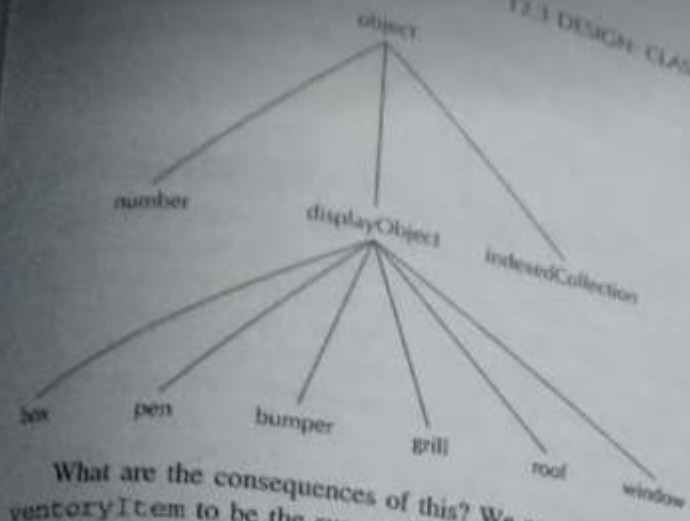


Figure 12.8 Example of `displayObject` Class Hierarchy

What are the consequences of this? We can choose either `displayObject` or `inventoryItem` to be the superclass of the other. Suppose we choose `displayObject`; then our class structure looks as shown in Figure 12.10. This seems to solve the problem. The display methods occur once—in `displayObject`—and the inventory control methods occur once—in `inventoryItem`. Unfortunately, this arrangement of the classes has a side effect: Some objects that are never displayed (e.g., engines and paint) now have the attributes of a displayed object, such as a display location. This means that they will respond to messages that are meaningless, which is a violation of the Security Principle. The alternative, placing `displayObject` under `inventoryItem`, is even worse since it means that objects such as `pens` and `boxes` will have attributes such as `weight`, `cost`, and `manufacturer`! Thus, we seem to be faced with a choice: either violate the Security Principle by making either `displayObject` or `inventoryItem` a subclass of the other or violate the Abstraction Principle by repeating in some of the classes the attributes of the others.

What is the source of this problem? In real life we often find that the same objects must be classified in several different ways. For example, a biologist might classify mammals as

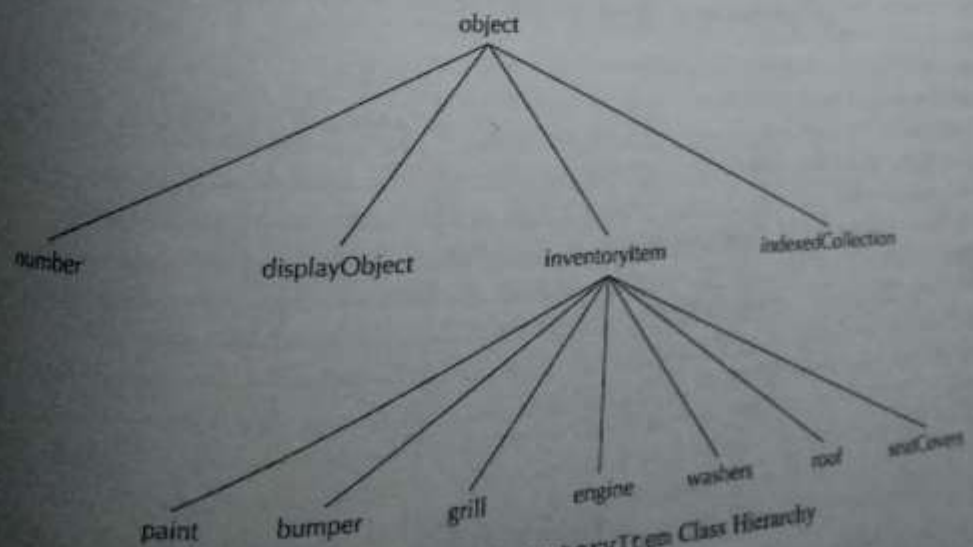


Figure 12.9 Example of `inventoryItem` Class Hierarchy

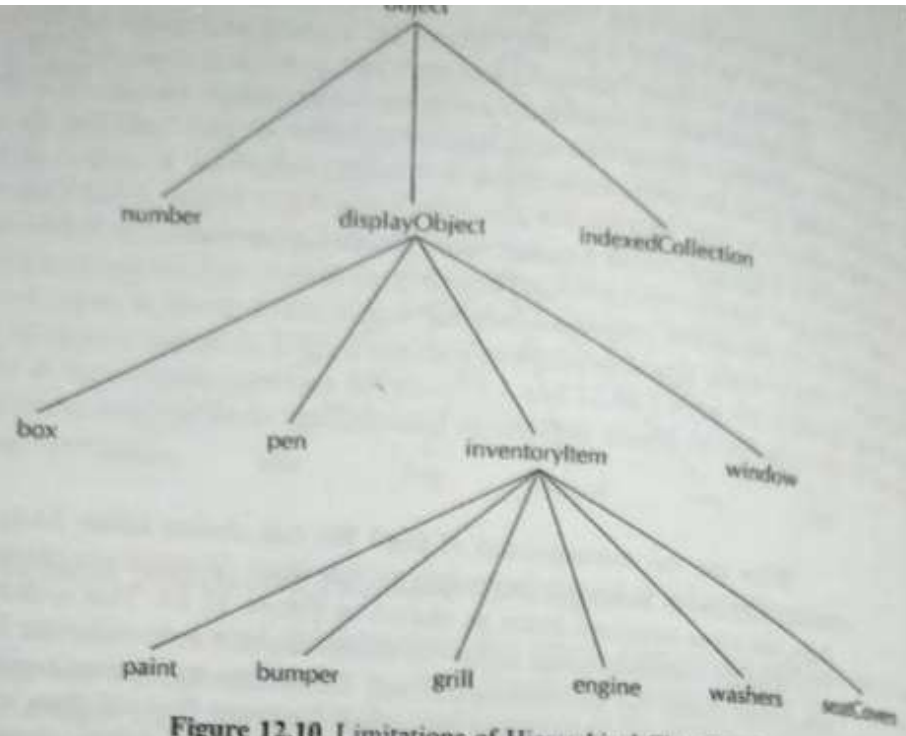


Figure 12.10 Limitations of Hierarchical Classification

primates, rodents, ruminants, and so forth. Someone interested in the uses of mammals may classify them as pets, beasts of burden, sources of food, pests, and so forth. Finally, a zoologist might classify them as North American, South American, African, and so forth. These are three *orthogonal* classifications; each of the classes cuts across the others at "right angles" (see Figure 12.11). (We have shown only two of the three dimensions.)

In summary, a hierarchical classification system, such as provided by Smalltalk, precludes orthogonal classification. This in turn forces the programmer to violate either the Secrets Principle or the Abstraction Principle. In essence, Smalltalk ignores the fact that the appropriate classification of a group of objects depends on the context in which those objects are viewed.

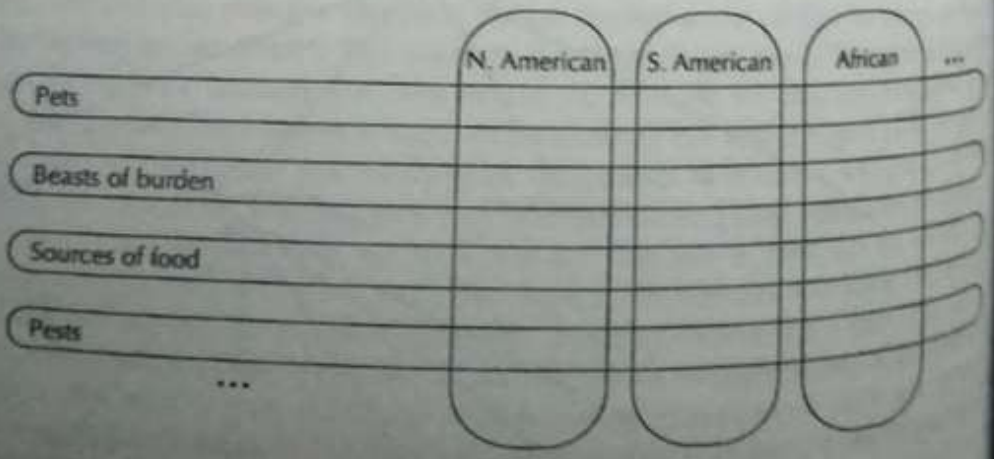


Figure 12.11 Example of Orthogonal Classification

LISP Recursive Interpreter and Reclamation

It is a fact the that first interpreter was a result of writing a universal function for LISP. A universal function is a function that can interpret any other function.

The recursive interpreter is written in LISP, although it could be written in any language with recursive procedures and the ability to implement linked lists. Since it is written in LISP, it makes use of the facilities of LISP, such as the list-processing operations 'car', 'cdr', and 'cons'. In particular, these operations are used to interpret 'car', 'cdr', and 'cons' operations in the program that we are interpreting. This might seem circular and pointless. In fact it is exactly analogous to the way floating-point operations were implemented in the pseudo-code interpreter in Chapter 1. There, floating-point operations in the implementation language (say, Pascal) were used to implement floating-point operations in the pseudo-code. Of course, if our implementation language hadn't had a floating-point capability, then these operations would have had to be implemented in terms of more basic operations. Similarly, if the implementation language for a LISP interpreter doesn't have list manipulation operations, it is necessary to implement these in terms of more basic operations. However, since we are using LISP as the implementation language, we can use list manipulation operations directly.

The LISP universal function is conventionally called 'eval' since it evaluates a LISP expression. In addition to the expression to be evaluated, 'eval' must have a second parameter, which is a data structure (a list of some sort) representing the context in which the evaluation is to be done. Recall that it is incomplete just to ask for the value of an expression; it is also necessary to specify the context of the evaluation (see Section 2.5, p. 78, and Section 6.1, p. 225). Hence, if *E* is any LISP expression (written in the *S*-expression notation) and *A* is a list representing a context, then

(eval 'E A) = V

where *V* is the value of *E* in that context. In other words, the result of evaluating '(eval 'E A)' is the same as the result of evaluating *E* in the context represented by *A*. Consider the following application (where we have assumed 'nil' represents the empty context):

```
(eval '(cons (quote A) (quote (B C D) )) nil)
  (A B C D)
```

The result agrees with the result of evaluating '(cons 'A '(B C D))' in the empty (or any other) context, which is '(A B C D)'.

Storage Reclamation

- What happens to *cons*'d pointers that are no longer in use?
- Explicit reclamation is the obvious / traditional way
  - C: malloc, calloc, realloc, free
  - C++: new, delete
  - Pascal: new, dispose
- Issues
  - Complicates programming
    - Requires the programmer to keep track of pointers
  - Violates security of the environment
    - Memory freed, but still referenced (dangling pointers)

Automatic Storage Reclamation

- It would be nice for the system to automatically 'reclaim' storage no longer used
- System can keep track of number of references to storage
  - When references decrease to 0, storage is returned to 'free-list'
- Advantage:
  - Storage reclaimed immediately as last reference is destroyed
- Disadvantage:
  - Cyclic structures (points to itself) cannot be reclaimed

Garbage Collection

- A different approach is garbage collection
  - Do not keep track of references to location
  - When last reference is destroyed, we still do not do anything, and leave the memory as garbage (unused, non-reusable storage, littering the memory)
  - Collect garbage if system runs out of storage
    - Mark all areas unused
    - Then examine all visible pointers and mark storage they point to as 'used'
    - Leftover is garbage, and can be put on free-list
  - This is called the *mark-and-sweep* method

Garbage Collection

- Advantages
  - Fast until runs out of memory
  - No additional memory is needed for tracking references
- Disadvantages
  - Garbage collection itself can be slow
    - If memory is large, and have many references
    - Must halt entire system, since all dynamic memory must be marked as unused first
- Java uses this approach

OOP concepts in c++ and java

Similar Constructors

Constructors work almost exactly the same way in C++ and Java. If you do not define a constructor for a class then it is allocated a default constructor. If we do define a constructor then we must use it. The only significant difference is that there is no copy constructor in Java as all objects of a class (i.e. non-primitive types) are passed by reference.

No Destructors!

There are no destructors in Java, even thought there is a `new` keyword, there is no corresponding `delete` keyword, as Java takes care of all of the memory management. Java has a special `finalize()` method that you can add to any class. The strange thing about this method is that you can use it for tidying up (printing a paper record for `Account` etc.), but you do not know when it will be called. In C++ this was easier, as the destructor is called when the object goes out of scope, but in Java the `finalize()` method will be called when the garbage collector destroys the object. We do not know when this will be called! As discussed previously, we can request that the garbage collector should run, but we do not know when it will run! You can help the garbage collector slightly by setting all your references to `null` when you no longer need them, e.g. `a = null;` - will send a hint to the garbage collector that the object is no longer being used.

Multiple Inheritance

C++ allows multiple inheritance - Java does not! As discussed previously in the C++ section of the notes, multiple inheritance is complex for the programmer, in that they must make complex decisions about the way that the data of the base class is to be inherited. In Java, we do not have multiple inheritance but we do have the use of Interfaces. Interfaces are a special kind of abstract class that have no data or implemented code. Interfaces might not sound too useful, but they allow a form of multiple inheritance, without having the associated difficulties of dealing with data. This replacement behaviour works very well and does not impact on development, once you get used to the change.

Nested Classes

In Java we have the facility to define a class inside of another class. We refer to this class as an **inner class**. The only issue worth mentioning with inner classes is that they have full access to the private data and methods of the class in which they are nested. Because of this, we do not need to pass state pointers to methods that need callback.

C++

```
class Foo {           // Declares class Foo
    int x = 0;        //  Private Member variable. It will
                      // be initialized to 0, if the
                      // constructor would not set it.
                      // (from C++11)
    public:
      Foo() : x(0)      //  Constructor for Foo; initializes
      {}                //  x to 0. If the initialiser were
                      //  omitted, the variable would
                      //  be initialised to the value that
                      // has been given at declaration of x.

      int bar(int i) { // Member function bar()
          return 3*i + x;
      }
};
```

```
Foo a;
// declares a to be a Foo object value,
// initialized using the default constructor.

// Another constructor can be used as
Foo a(args);
// or (C++11):
Foo a{args};
```

Java

```
class Foo {               // Defines class Foo
    private int x;        // Member variable, normally declared
                          // as private to enforce encapsulation
                          // initialized to 0 by default

    public Foo() {        // Constructor for Foo
    }                     // no-arg constructor supplied by
default

    public int bar(int i) {        // Member method bar()
        return 3*i + x;
    }
}
```

```
Foo a = new Foo();
// declares a to be a reference to a new Foo object
// initialized using the default constructor

// Another constructor can be used as
Foo a = new Foo(args);
```

C++ supports operator overloading.

User defined function in LISP

Use defun to define your own functions in LISP. Defun requires you to provide three things. The first is the name of the function, the second is a list of parameters for the function, and the third is the body of the function -- i.e. LISP instructions that tell the interpreter what to do when the function is called.

Schematically then, a function definition looks like this:

```
(defun <name> <parameter-list> <body>)
```

Here is an example of a function to calculate the square of a number. It is short enough to be typed directly at the interpreter's prompt:

```
>(defun square (x)
   (* x x))
SQUARE

>(square 2)
4

>(square 1.4142158)
2.0000063289696399
```

The macro named **defun** is used for defining functions. The **defun** macro needs three arguments −

- Name of the function
- Parameters of the function
- Body of the function

Syntax for defun is −

```
(defun name (parameter-list) "Optional documentation string." body)
```

Please note that −

- You can provide an empty list as parameters, which means the function takes no arguments, the list is empty, written as ().
- LISP also allows optional, multiple, and keyword arguments.
- The documentation string describes the purpose of the function. It is associated with the name of the function and can be obtained using the **documentation** function.
- The body of the function may consist of any number of Lisp expressions.
- The value of the last expression in the body is returned as the value of the function.
- You can also return a value from the function using the **return-from** special operator.

Contour Diagram

Subprogram Names Are Global in Scope

Clearly, the above argument cannot apply to the names of the subprograms themselves; if the name of a subprogram were only visible within that subprogram, then it would be impossible for anyone else to call that subprogram.

Instead, FORTRAN specifies that subprogram names are visible throughout the program; they are thus said to be *global*. In contrast we say that variable names are *local* to the subprograms in which they are declared. Thus, the names fall into two broad classes, depending on their visibility. This property is called the *scope* of a name; that is, subprogram names have global scope and variable names have local scope. The *scope* of a binding of a name is defined as that region of the program over which that binding is visible. Consider Figure 2.8. In this program the scope of the binding of X marked (*) is the main program and the scope of the binding of X marked (**) is the subroutine R. Similarly, the scope of the binding of Y (**) is R and the scope of the binding (***) is S. The scopes of the two uses of N as formal parameters are the bodies of the corresponding subroutines. Finally, the scope of the declarations of the subroutines R and S is the entire program.

These relationships can be depicted in a *contour diagram*, such as that in Figure 2.9. These diagrams (which were invented by John B. Johnston[8]) should be interpreted as though the boxes were made of one-way mirrors that allow us

```
C       MAIN PROGRAM
        INTEGER X                    (*)
          :
        CALL R(2)
          :
        CALL S(X)
          :
        END

        SUBROUTINE R(N)
        REAL X, Y                    (**)
          :
        CALL S(Y)
          :
        END

        SUBROUTINE S(N)
        INTEGER Y                    (***)
          :
        END
```

Figure 2.8 Examples of Variable Scopes in FORTRAN

8 See Johnston (1971).

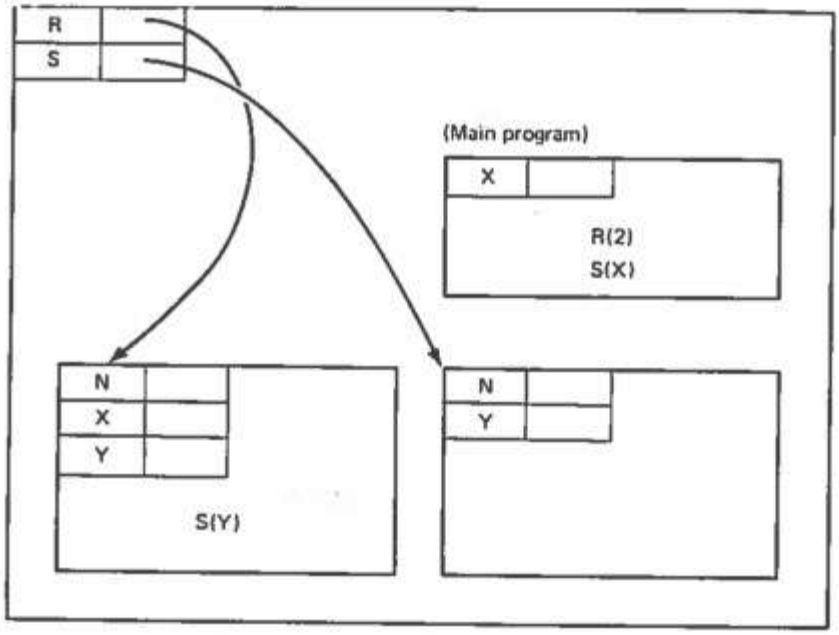


Figure 2.9 Contour Diagram Showing Scopes of Variables

to look out of a box but not into one. The arrows on the diagram show, for example, that R can see the bindings of S and its own Y, but that it cannot see the Y in S or the X in the main program. Contour diagrams are a valuable aid to visualizing scopes, particularly in languages with more complicated scope rules than FORTRAN. We will use them again in later chapters.

EXERCISE 2-27: Write in skeleton form a FORTRAN program involving at least three subprograms. Draw a contour diagram that shows the visibility relations between the bindings and uses of names.

Block and Scope and Contour

Blocks Define Nested Scopes

In FORTRAN we saw that environments are composed of scopes nested in two levels. All subprograms are bound in the outer (global) scope and all (subprogram-local) variables are bound in inner scopes, one for each subprogram (see Figure 2.9). Although COMMON blocks are effectively bound at the global level (since they are visible to all subprograms), in fact they must be redeclared in each subprogram. Algol-60 avoids this redeclaration by allowing the programmer to define any number of scopes nested to any depth; this is accomplished with a *block*:

begin declarations; statements **end**

This defines a scope that extends from the **begin** to the **end**. This is the scope of the names bound in the declarations immediately following the **begin**; therefore, these names are visible to all of the statements in the block. Since these statements may themselves be blocks, we can see that the scopes can be nested.

Contour diagrams are often helpful in visualizing name structures. Let's compare the program in Figure 3.1 with the contour diagram in Figure 3.2 to be sure that we understand it. Remember that the rule for contour diagrams is that we can look out of a box but we can't look into one. Figure 3.3 shows an outline of a more complicated Algol program; its contour diagram is in Figure 3.4.

Notice that the contours are suggested by the *scoping lines* we have drawn to the left of the program in Figure 3.3. Contour diagrams originated by completing scoping lines into boxes. We can see that in addition to blocks, procedure declarations also introduce a level of nesting since the formal parameters are local to the procedure. We can also see where the name "contour diagram" came from; the diagrams are suggestive of contour maps.

We have said that the purpose of name structures is to organize the name space. Why is this important? Virtually everything a programmer deals with in a program is named. Therefore, as programs become larger and larger, there will be more and more names for the programmer to keep track of, which can make understanding and maintaining the program very difficult. Another way to say this is that the *context* that programmers must keep in their heads is too large; too many names are visible. Therefore, the goal of name structures is to limit the context with which the programmer must deal at any given time. Name structures do this by restricting the visibility of names to particular parts of a program, in the case of block structure, to the block in which the name is declared. For example, in the program in Figure 3.3, the variable 'val' is only needed for the

```
begin
  real x, y;
  real procedure cosh(x); real x;
    cosh := (exp(x)+exp(-x))/2;

  procedure f(y,z);
    integer y, z;
    begin real array A[1:y];
      :
    end
  :
  begin integer array Count [0:99];
    :
  end
  :
end
```

Figure 3.3 Nested Environments

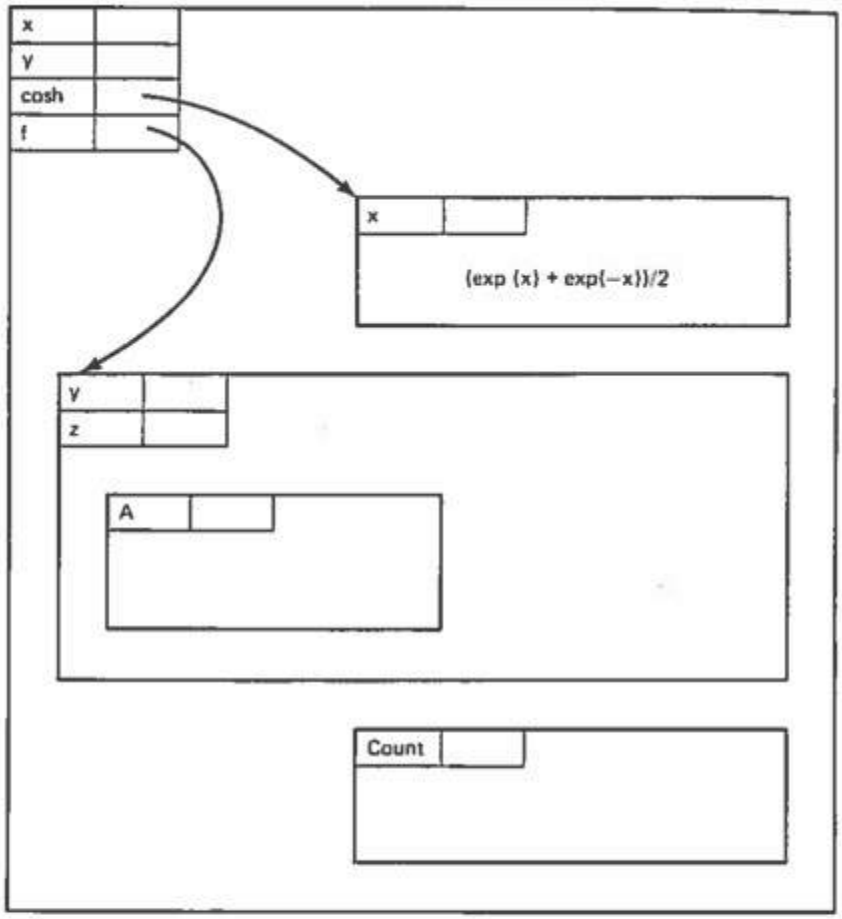


Figure 3.4 Contour Diagram of Nested Scopes

two statements in the body of the first **for**-loop. Therefore, it is declared in the block that forms the body. We can see from this example that it would be very inconvenient if the variables declared in the outer blocks (N, Data, sum, avg, and i) were not visible in the inner block. For this reason, an inner block *implicitly inherits* access to all of the variables accessible in its immediately surrounding block; this is what's shown by the contour diagrams. The names declared in a block are called *local* to that block; those declared in surrounding blocks are called *nonlocal*. The names declared in the outermost block are called *global* because they are visible to the entire program.

Lambda expression

C.4.3 A `lambda` Expression: Useful Anonymity

`lambda` is the symbol for an anonymous function, a function without a name. Every time you use an anonymous function, you need to include its whole body.

Thus,

```
(lambda (arg) (/ arg 50))
```

is a function that returns the value resulting from dividing whatever is passed to it as `arg` by 50.

Earlier, for example, we had a function `multiply-by-seven`; it multiplied its argument by 7. This function is similar, except it divides its argument by 50; and, it has no name. The anonymous equivalent of `multiply-by-seven` is:

```
(lambda (number) (* 7 number))
```

(See The `defun` Macro.)

If we want to multiply 3 by 7, we can write:

```
(multiply-by-seven 3)
 \_____________/ ^
        |        |
     function  argument
```

This expression returns 21.

Similarly, we can write:

```
((lambda (number) (* 7 number)) 3)
 \_____________________________/ ^
                |                 |
      anonymous function      argument
```

If we want to divide 100 by 50, we can write:

```
((lambda (arg) (/ arg 50)) 100)
 \_____________________/  \_/
             |             |
   anonymous function   argument
```

This expression returns 2. The 100 is passed to the function, which divides that number by 50.

Zero-one-infinity principle

Zero one infinity rule

From Wikipedia, the free encyclopedia

The **Zero one or infinity (ZOI) rule** is a rule of thumb in software design proposed[*when?*] by early computing pioneer Willem van der Poel.[1] It argues that arbitrary limits on the number of instances of a particular entity[*jargon*] should not be allowed. Specifically, an entity should either be forbidden entirely, only one should be allowed, or any number of them should be allowed.[2] Although various factors outside that particular software could limit this number in practice, it should not be the software itself that puts a hard limit on the number of instances of the entity.

"Allow none of foo, one of foo, or any number of foo."

"The only reasonable numbers are zero, one and infinity." -- Bruce J. MacLennan

Examples of this rule may be found in the structure of many file systems' directories (also known as folders):

- 0 – The topmost directory has zero parent directories; that is, there is no directory that contains the topmost directory.
- 1 – Each subdirectory has exactly one parent directory (not including shortcuts to the directory's location; while such files may have similar icons to the icons of the destination directories, they are not directories at all).
- ∞ (infinity) – Each directory, whether the topmost directory or any of its subdirectories, according to the file system's rules, may contain any number of files or subdirectories. Practical limits to this number are caused by other factors, such as space available on storage media and how well the computer's operating system is maintained.[*citation needed*]

Note that violations of this rule of thumb do exist: for example, some file systems impose a limit of 65,536 (i.e. 2^{16}) files to a directory.[3]

- indentifiers allowed to be any length (zero one infinity enforcement)
- no longer had restriction on number of dimensions to arrays (zero one infinity enforcement)

Arbitrary Restrictions Are Eliminated

Algol adheres to the Zero-One-Infinity Principle in many of its design decisions. For example, there is no limit on the number of characters in an identifier (FORTRAN allowed at most six). It also eliminates many other restrictions, for example, any expression is allowed as a subscript, including other array elements:

```
A[2 × B[1] − 1]
Count [if i > 100 then 100 else i]
```

Although these facilities are not often needed, they reduce the number of special cases the programmer must learn and they support the idea that "anything you think you ought to be able to do, you will be able to do." All of these features contribute to Algol's reputation as a general, regular, elegant, orthogonal language.

s-expression in lisp

11.1 Introduction to Symbol Expressions

The S-expression

The syntactic elements of the Lisp programming language are *symbolic expressions*, also known as *s-expressions*. Both programs and data are represented as s-expressions: an s-expression may be either an *atom* or a *list*. Lisp atoms are the basic syntactic units of the language and include both numbers and symbols. Symbolic atoms are composed of letters, numbers, and the non-alphanumeric characters.

An *s-expression* is defined recursively:

An *atom* is an s-expression.

If s_1, s_2, ..., s_n are s-expressions, then so is the list $(s_1\ s_2\ \ldots\ s_n)$.

A *list* is a non-atomic s-expression.

A form is an s-expression that is intended to be evaluated. If it is a list, the first element is treated as the function name and the subsequent elements are evaluated to obtain the function arguments.

E.3 Garbage Collection

When a program creates a list or the user defines a new function (such as by loading a library), that data is placed in normal storage. If normal storage runs low, then Emacs asks the operating system to allocate more memory. Different types of Lisp objects, such as symbols, cons cells, small vectors, markers, etc., are segregated in distinct blocks in memory. (Large vectors, long strings, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object; small strings are packed into blocks of 8k bytes, and small vectors are packed into blocks of 4k bytes).

Beyond the basic vector, a lot of objects like window, buffer, and frame are managed as if they were vectors. The corresponding C data structures include the `union vectorlike_header` field whose `size` member contains the subtype enumerated by `enum pvec_type` and an information about how many `Lisp_Object` fields this structure contains and what the size of the rest data is. This information is needed to calculate the memory footprint of an object, and used by the vector allocation code while iterating over the vector blocks.

It is quite common to use some storage for a while, then release it by (for example) killing a buffer or deleting the last pointer to an object. Emacs provides a garbage collector to reclaim this abandoned storage. The garbage collector operates by finding and marking all Lisp objects that are still accessible to Lisp programs. To begin with, it assumes all the symbols, their values and associated function definitions, and any data presently on the stack, are accessible. Any objects that can be reached indirectly through other accessible objects are also accessible.

When marking is finished, all objects still unmarked are garbage. No matter what the Lisp program or the user does, it is impossible to refer to them, since there is no longer a way to reach them. Their space might as well be reused, since no one will miss them. The second (sweep) phase of the garbage collector arranges to reuse them.

The sweep phase puts unused cons cells onto a free list for future allocation; likewise for symbols and markers. It compacts the accessible strings so they occupy fewer 8k blocks; then it frees the other 8k blocks. Unreachable vectors from vector blocks are coalesced to create largest possible free areas; if a free area spans a complete 4k block, that block is freed. Otherwise, the free area is recorded in a free list array, where each entry corresponds to a free list of areas of the same size. Large vectors, buffers, and other large objects are allocated and freed individually.

First Generation

Characteristics of First-Generation Programming Languages

In this chapter we have used FORTRAN (specifically, FORTRAN IV) to illustrate the characteristics of *first-generation* programming languages. We now pause to summarize those characteristics.

In general, we've seen that the structures of first-generation languages are based on the structures of the computers of the early 1960s. This is natural, since the only experience people had in programming was in programming these machines.

This machine orientation is especially apparent in first-generation *control structures*. For example, FORTRAN's control structures correspond almost one for one with the branch instructions of the IBM 704. First-generation conditional instructions are nonnested (unlike those in later languages), and first-generation languages depend heavily on the GOTO for building any but the simplest control structures. One exception to this rule is the definite iteration statement (e.g., FORTRAN's DO-loop), which is hierarchical in first-generation languages. Recursive procedures are not permitted in most first-generation languages (BASIC is an exception), and there is generally only one parameter passing mode (typically, pass by reference).

The machine orientation of first-generation languages can also be seen in the types of *data structures* provided, which are patterned after the layout of memory on the computers available around 1960. Thus, the data structure *primitives* found in first-generation languages are fixed and floating-point numbers of various precisions, characters, and logical values—just the kinds of values manipulated by the instructions on these computers. The data structure *constructors* are arrays and, in business-oriented languages, records, which are the ways storage was commonly organized. As with control structures, first-generation languages provide little facility for hierarchical data organization (an exception is COBOL's record structure). That is, data structures cannot be nested. Finally, many first-generation languages are characterized by a relatively weak type system; that is,it is easy to subvert the type system or do representation-dependent programming.

Hierarchical structure is also absent from first-generation *name structures*, with disjoint scopes being the rule. Furthermore, variable names are bound directly and statically to memory locations since there is no dynamic memory management.

Finally, the *syntactic structures* of first-generation languages are characterized by a card-oriented, linear arrangement of statements patterned after assembly languages. Further, most of these languages had numeric statement labels that are suggestive of machine addresses. First-generation languages go significantly beyond assembly languages, however, in their provision of algebraic notation. Their usual lexical conventions are to ignore blanks and to recognize keywords in context.

In summary, the salient characteristics of the first generation are machine orientation and linear structures. We will see in the next chapter that the second generation makes important moves in the directions of application orientation and hierarchical structure.