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.

GOTO Statements in Fortran

GO TO (Assigned)

The **assigned** `GO TO` statement branches to a statement label identified by the assigned label value of a variable.

`GO TO` *i* `[[,](` *s* `[,` *s* `])]`

Parameter	Description
i	Integer variable name
s	Statement label of an executable statement

Description

Execution proceeds as follows:

1. At the time an assigned `GO TO` statement is executed, the variable **i** must have been assigned the label value of an executable statement in the same program unit as the assigned `GO TO` statement.
2. If an assigned `GO TO` statement is executed, control transfers to a statement identified by **i**.
3. If a list of statement labels is present, the statement label assigned to **i** must be one of the labels in the list.

Restrictions

i must be assigned by an `ASSIGN` statement in the same program unit as the `GO TO` statement.

i must be `INTEGER*4` or `INTEGER*8`, not `INTEGER*2`.

s must be in the same program unit as the `GO TO` statement.

The same statement label can appear more than once in a `GO TO` statement.

The statement control jumps to must be **executable**, not `DATA`, `ENTRY`, `FORMAT`, or `INCLUDE`.

Control cannot jump into a `DO`, `IF`, `ELSE IF`, or `ELSE` block from outside the block.

Example

Example: Assigned `GO TO`:

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

GO TO (Computed)

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.

```
GO TO ( s [, s ])[,] e
```

Parameter	Description
s	Statement label of an **executable** statement
e	Expression of type integer or real

Description

Execution proceeds as follows:

1. *e* is evaluated first. It is converted to integer, if required.
2. If 1 £ *e* £ *n*, where *n* is the number of statement labels specified, then the eth label is selected from the specified list and control is transferred to it.
3. If the value of **e** is outside the range, that is, **e** < 1 or **e** > **n**, then the computed GO TO statement serves as a CONTINUE statement.

Restrictions

s must be in the same program unit as the GO TO statement.

The same statement label can appear more than once in a GO TO statement.

The statement control jumps to must be **executable**, not DATA , ENTRY , FORMAT , or INCLUDE .

Control cannot jump into a DO , IF , ELSE IF , or ELSE block from outside the block.

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.

Dynamic Chain of Activation Record

Subprograms Are Implemented Using Activation Records

In this section we investigate the way subprograms (subroutines and functions) are implemented. What happens when a subprogram is invoked? Clearly, we must transmit the parameters to the subprogram, which is the first step. This may be done by reference or by value (the first half of the value-result process).

It may seem that the next step is to enter the subprogram, but we must do something else first. If we entered now, there would be no way to get back to the caller because a subprogram can be called from many different callers and from many different places within one caller (as we saw in the DIST example). Therefore, there is not a unique place to which the callee should return when it has finished. To put it another way, it is necessary to tell the callee who its caller is; then, when the callee executes its RETURN statement it will know to whom to return.

There is one other issue we must address: saving the state of the caller. As you probably know, most computers have a number of *registers* that can be used for high-speed temporary storage. Since subprograms may be separately compiled, it is not usually possible to know the registers that another subprogram uses. Therefore, when one subprogram calls another, it is necessary for one or the other to preserve the content of the registers in a private area of memory. The content of the registers can then be restored when the caller gets control back from the callee.

We summarize these ideas as follows: When one subprogram calls another, the *state* of the caller must be preserved before the callee is entered and must be restored after the callee returns. By the state of the caller, we mean all of the information that characterizes the state of the computation in progress. This includes the contents of all of the variables, the contents of any registers in use, and the current point of execution (as indicated by the IP, or instruction pointer register). Of course, any information that is already stored in memory locations private to the caller is already saved and need not be saved again. All of the other information must be stored in a caller-private data area before the callee is entered. This data area is often called an *activation record* because it holds all the information relevant to one *activation* of a subprogram. A subprogram is *active* if it's been called but hasn't yet returned. In a nonrecursive language such as FORTRAN, there is one activation record for each subprogram (activation records for recursive subprograms are discussed in Chapter 3). The concept of an activation record is very important and will be discussed repeatedly in the following chapters.

The activation record serves a number of useful functions. For example, we have said that the caller must *transmit* the actual parameters to the callee. This means that the actual parameters (either their values or their references) must be placed in a location where the callee knows to find them. Where should this be? The callee's activation record is often a good choice (although there are others, such as registers). Thus, a subprogram's state includes its current parameters.

Since the activation record contains all of the information needed to restart a subprogram (its IP, register values, etc.), it is convenient to think of the activation record as a repository for all information relevent to the subprogram. We said earlier that the callee must be passed some sort of reference to the caller so that it will know which caller to resume when it returns. A very convenient way to do this is to transmit to the callee a pointer to the caller's activation record. The callee then has all of the information required to resume its caller. For example, the return address for the return jump to the caller can be obtained by the callee from the caller's activation record.

How are we to transmit to the callee the pointer to the caller's activation record? The simplest method is to store the pointer in the callee's activation record. This pointer from a callee's activation record to its caller's activation record is called a *dynamic link*. The *dynamic chain* is the sequence of dynamic links that reach back from each callee to its caller. The dynamic chain begins at the currently active subprogram (i.e., the one now in control) and terminates at the main program. See Figure 2.3 for an example. We show the situation in which the main program has called *S*, *S* has called *T*, *T* has called *F*, and *F* is still active.

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.

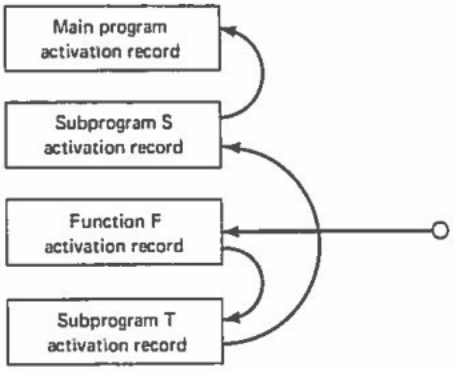


Figure 2.3 The Dynamic Chain of Activation Records

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)

It is not particularly important what format is used for this information, although certain layouts may be particularly efficient on certain machines (Figure 2.4).

We can make these ideas a little more specific by looking at the code for each of the steps in a CALL and a RETURN. In order to do this, we have to introduce some notation. Rather than introduce the assembly language for either a real or made-up machine, we use a conventional high-level language syntax. We must be careful to use statements that are very simple so that they can be implemented with one or two instructions on most machines. First, we use the notation 'M[*k*]' to represent the memory location with the address *k*. For example,

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

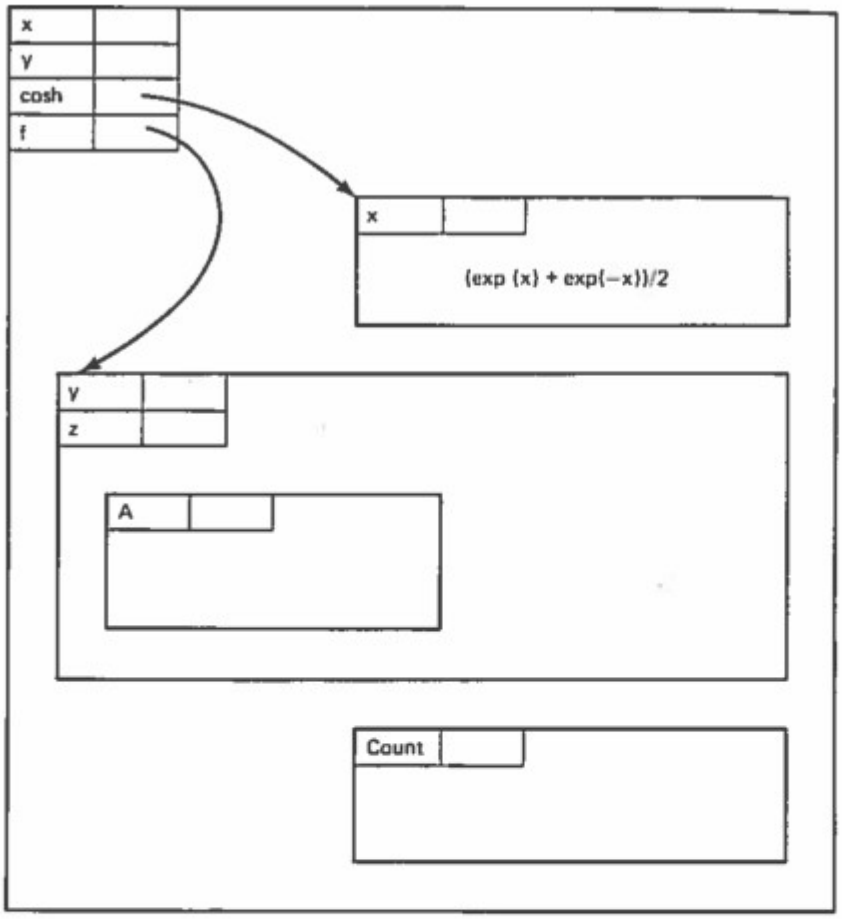


Figure 3.4 Contour Diagram of Nested Scopes

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.

In computer science, **Backus–Naur form** or **Backus normal form** (**BNF**) is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols. They are applied wherever exact descriptions of languages are needed: for instance, in official language specifications, in manuals, and in textbooks on programming language theory.

Many extensions and variants of the original Backus–Naur notation are used; some are exactly defined, including extended Backus–Naur form (EBNF) and augmented Backus–Naur form (ABNF).

In computer science, **extended Backus–Naur form** (**EBNF**) is a family of metasyntax notations, any of which can be used to express a context-free grammar. EBNF is used to make a formal description of a formal language such as a computer programming language. They are extensions of the basic Backus–Naur form (BNF) metasyntax notation.

The earliest EBNF was developed by Niklaus Wirth incorporating some of the concepts (with a different syntax and notation) from Wirth syntax notation. However, many variants of EBNF are in use. The International Organization for Standardization adopted an EBNF standard (ISO/IEC 14977) in 1996. However, according to Zaytsev this standard "only ended up adding yet another three dialects to the chaos" and, after noting its lack of success, also notes that the ISO EBNF is not even used in all ISO standards. Wheeler argues against using the ISO standard when using an EBNF, and recommends considering alternative EBNF notations such as the one from the W3C Extensible Markup Language (XML) 1.0 (Fifth Edition).

Introduction [edit]

A BNF specification is a set of derivation rules, written as

```
<symbol> ::= __expression__
```

where <symbol>[5] is a *nonterminal*, and the __expression__ consists of one or more sequences of symbols; more sequences are separated by the vertical bar "|", indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are *terminals*. On the other hand, symbols that appear on a left side are *non-terminals* and are always enclosed between the pair <>.[5]

The "::=" means that the symbol on the left must be replaced with the expression on the right.

<adding operator> ::= + |-
<multiplying operator> ::= ×|/|↑
<primary> ::= <unsigned number> | <variable> | <function designator> |
(<arithmetic expression>)
<factor> ::= <primary> | <factor> ↑ <primary>
<simple arithmetic expression> ::= <term> | <adding operator><term> |
<simple arithmetic expression><adding operator><term>
<if clause> ::= if <Boolean expression> then
<arithmetic expression> ::= <simple arithmetic expression> |
<if clause><simple arithmetic expression> else <arithmetic expression>

Figure 3: The definition of arithmetic expressions in the ALGOL 60 report using the BNF. (From: Naur, ed., Report on the Algorithmic Language ALGOL 60, p. 17)

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.

S-Expression

In Common Lisp, a s-exp would be like:

s-exp : (op s-exp1 s-exp2 ...)
op: a function | a macro | a special form

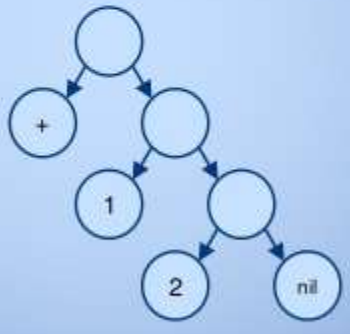
And interestingly, a s-exp could be 'made' like this:

(cons 1 2) => (1 . 2)
(cons 1 (cons 2 nil)) => (1 . (2 . nil)) => (1 2)
(cons '+ (cons 1 (cons 2 nil))) => (+ 1 2)
(eval (cons '+ (cons 1 (cons 2 nil)))) => 3

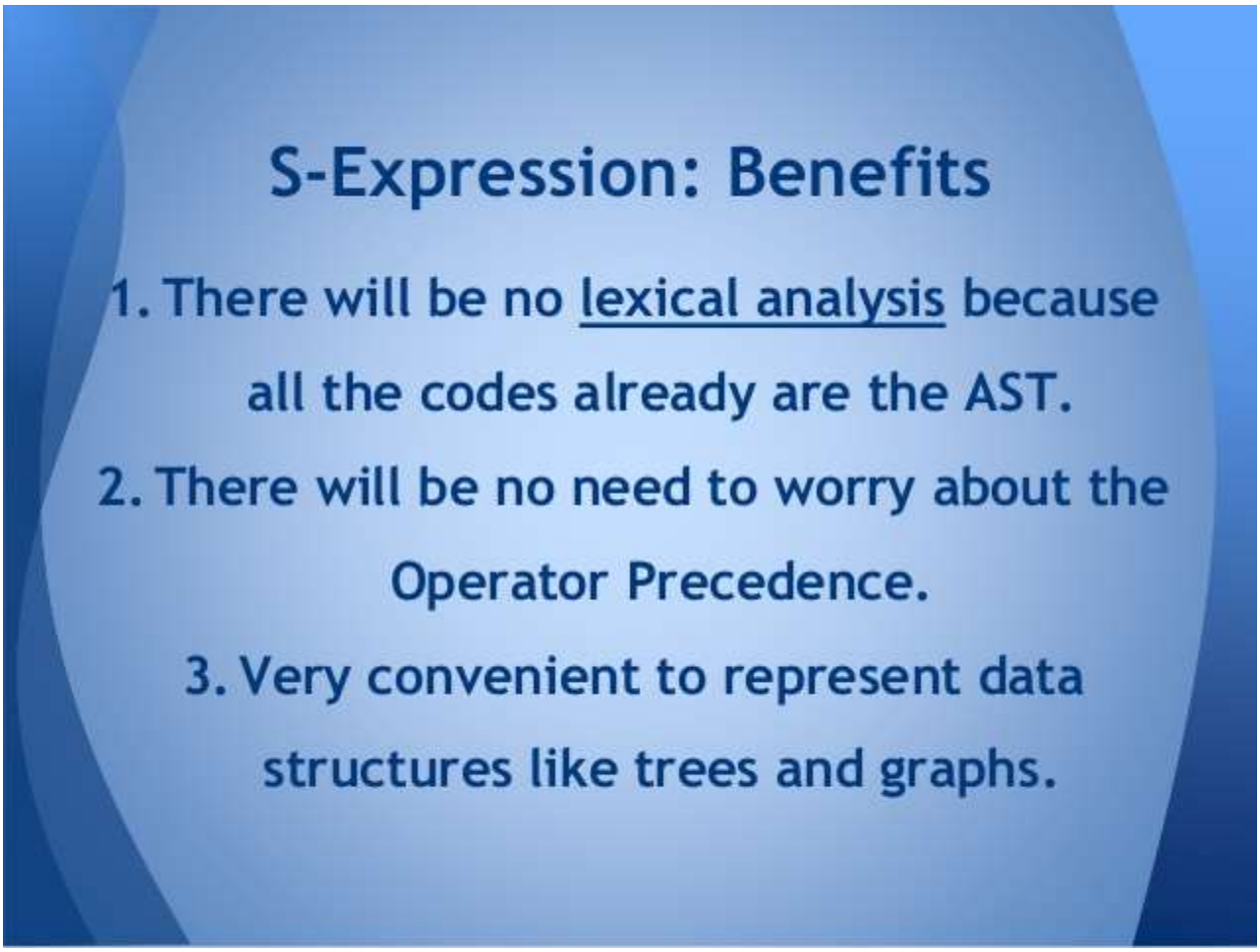
S-Expression

A s-exp looks like a linked list but actually a tree.

(car (list '+ 1 2)) => '+
(cdr (list '+ 1 2)) => (1 2)



Writing s-expressions is actually writing the **Abstract Syntax Tree(AST)**.



LISP Control Structures

Lisp originally had very few control structures, but many more were added during the language's evolution. (Lisp's original conditional operator, `cond`, is the precursor to later `if-then-else` structures.)

Sequencing

Evaluating forms in the order they appear is the most common way control passes from one form to another. In some contexts, such as in a function body, this happens automatically. Elsewhere you must use a control structure construct to do this: `progn`, the simplest control construct of Lisp.

A `progn` special form looks like this:

```
(progn a b c ...)
```

and it says to execute the forms *a*, *b*, *c* and so on, in that order. These forms are called the body of the `progn` form. The value of the last form in the body becomes the value of the entire `progn`.

Conditionals

Conditional control structures choose among alternatives. Emacs Lisp has two conditional forms: `if`, which is much the same as in other languages, and `cond`, which is a generalized case statement.

Special Form: **if** *condition then-form else-forms...*

`if` chooses between the *then-form* and the *else-forms* based on the value of *condition*. If the evaluated *condition* is non-`nil`, *then-form* is evaluated and the result returned. Otherwise, the *else-forms* are evaluated in textual order, and the value of the last one is returned. (The *else* part of `if` is an example of an implicit `progn`. See section Sequencing.)

If *condition* has the value `nil`, and no *else-forms* are given, `if` returns `nil`.

`if` is a special form because the branch that is not selected is never evaluated--it is ignored. Thus, in the example below, `true` is not printed because `print` is never called.

```
(if nil
    (print 'true)
  'very-false)
=> very-false
```

Special Form: **cond** *clause...*

`cond` chooses among an arbitrary number of alternatives. Each *clause* in the `cond` must be a list. The CAR of this list is the *condition*; the remaining elements, if any, the *body-forms*. Thus, a clause looks like this:

```
(condition body-forms...)
```

`cond` tries the clauses in textual order, by evaluating the *condition* of each clause. If the value of *condition* is non-`nil`, the clause "succeeds"; then `cond` evaluates its *body-forms*, and the value of the last of *body-forms* becomes the value of the `cond`. The remaining clauses are ignored.

If the value of *condition* is `nil`, the clause "fails", so the `cond` moves on to the following clause, trying its *condition*.

If every *condition* evaluates to `nil`, so that every clause fails, `cond` returns `nil`.

The following example has four clauses, which test for the cases where the value of `x` is a number, string, buffer and symbol, respectively:

```
(cond ((numberp x) x)
      ((stringp x) x)
      ((bufferp x)
       (setq temporary-hack x) ; multiple body-forms
       (buffer-name x))        ; in one clause
      ((symbolp x) (symbol-value x)))
```

Often we want to execute the last clause whenever none of the previous clauses was successful. To do this, we use `t` as the *condition* of the last clause, like this: `(t body-forms)`. The form `t` evaluates to `t`, which is never `nil`, so this clause never fails, provided the `cond` gets to it at all.

For example,

```
(cond ((eq a 'hack) 'foo)
      (t "default"))
=> "default"
```

This expression is a `cond` which returns `foo` if the value of `a` is 1, and returns the string `"default"` otherwise.

Object Representation

Much of Smalltalk's implementation can be derived by application of the Abstraction and Information Hiding Principles. For example, the representation of an object must contain just that information that varies from object to object; information that is the same over a class of objects is stored in the representation of that class. What is the information that varies between the instances of a class? It is just the *instance variables*. The information stored with the class includes the class methods and instance methods.

Notice, however, that we will not be able to access the information stored with the class of an object unless we know what the class of that object is. Therefore, the representation of an object must contain some indication of the class to which the object belongs. There are many ways to do this, and several have been used by the various Smalltalk implementations. The simplest is to include in the representation of the object a pointer to the data structure representing the class.

Let's consider the example shown in Figure 12.9, which shows the representation of two boxes, B1 and B2. To keep the figure clear, we have abbreviated or omitted some of the component objects. For example, the representation of the object '500@200' is shown, but the representation of '500@600' is abbreviated. Also, we have shown the names of the instance variables, although there is no reason actually to store them in the representation of objects. Finally, the representation of class objects is omitted because this topic is discussed next.

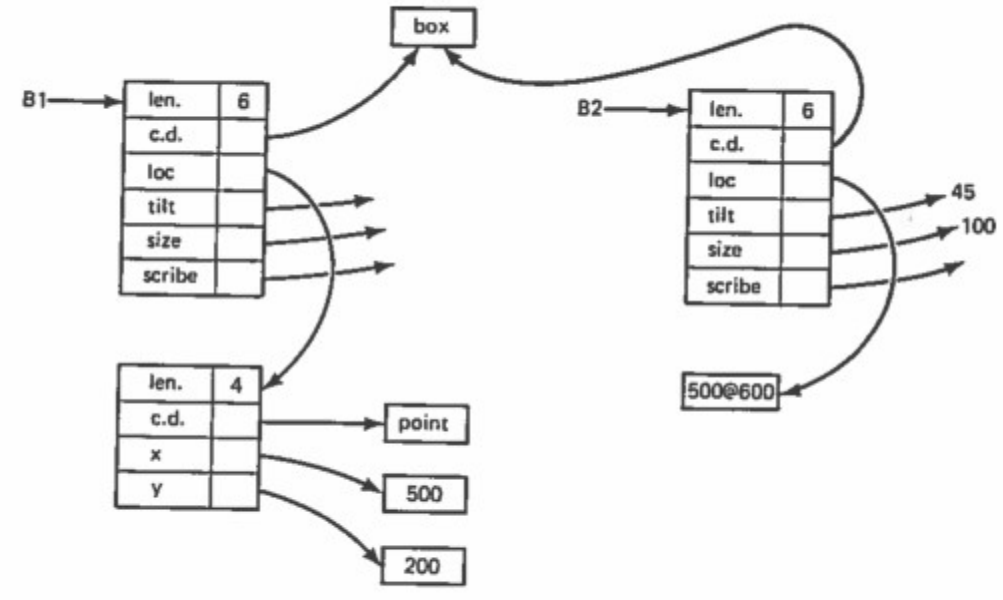


Figure 12.9 Representation of Objects

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

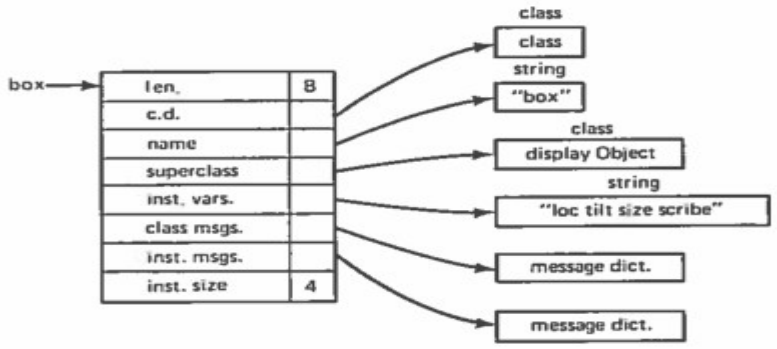


Figure 12.10 Representation of Class Object

There Are Three Forms of Message Template

You have seen that messages are essentially procedure invocations, although the formats allowed for messages are a little more flexible. In most languages parameters 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.

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.

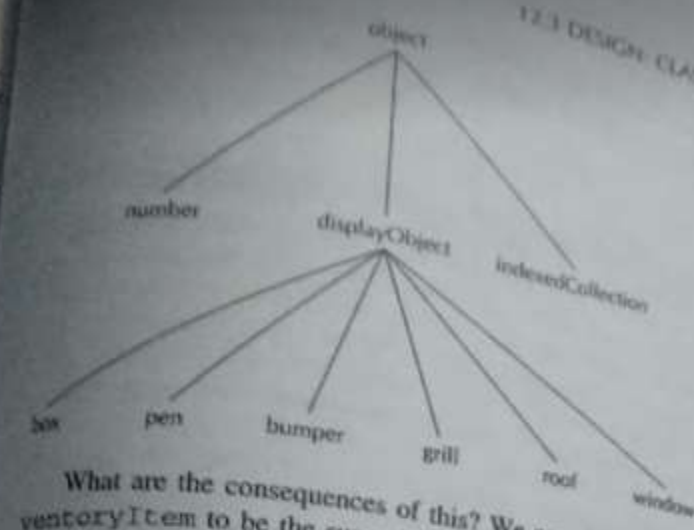


Figure 12.8 Example of displayObject Class Hierarchy

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

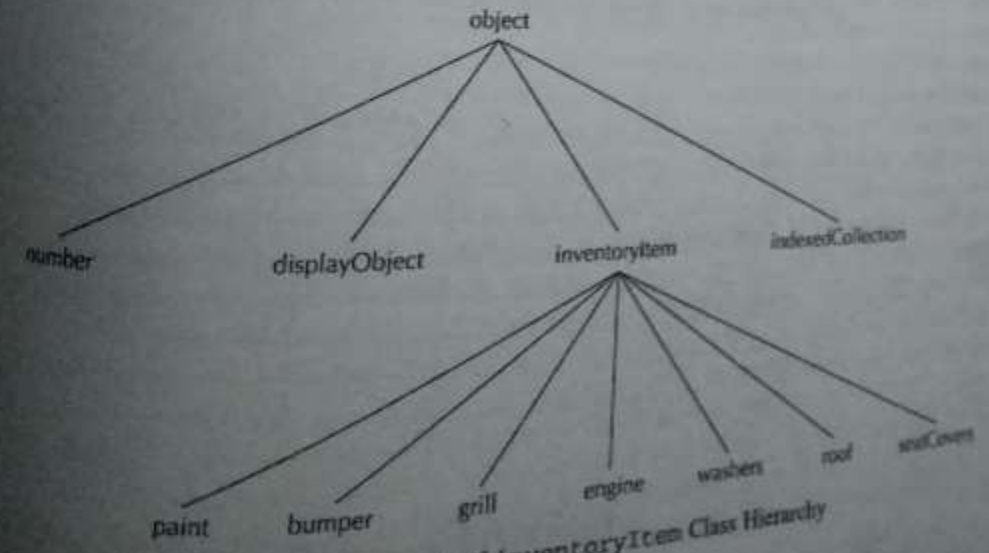


Figure 12.9 Example of `inventoryItem` Class Hierarchy

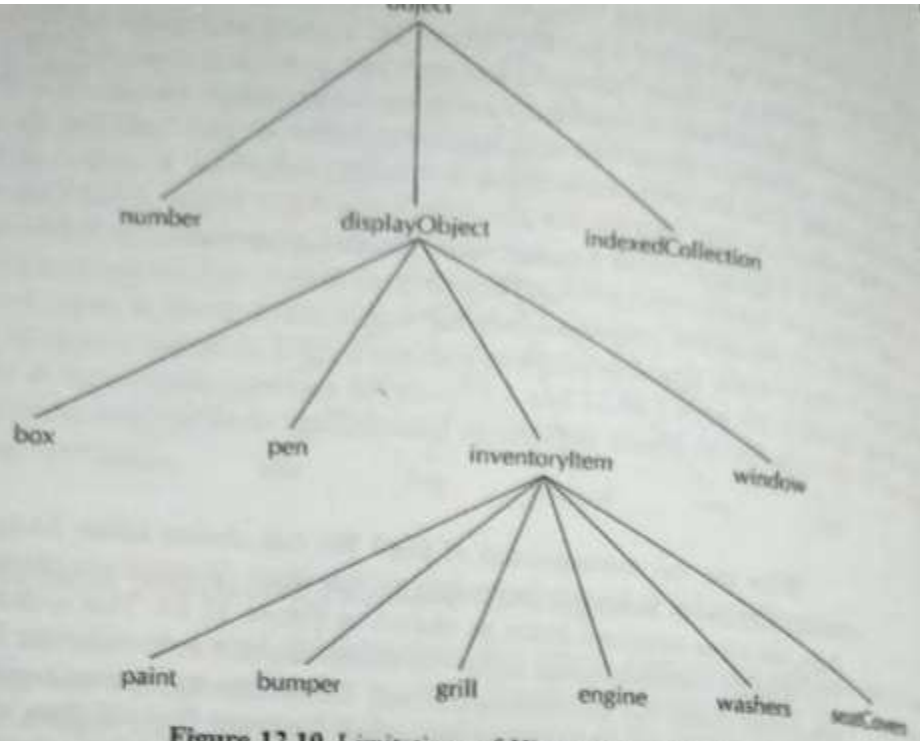


Figure 12.10 Limitations of Hierarchical Classification

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 Secrecy 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.

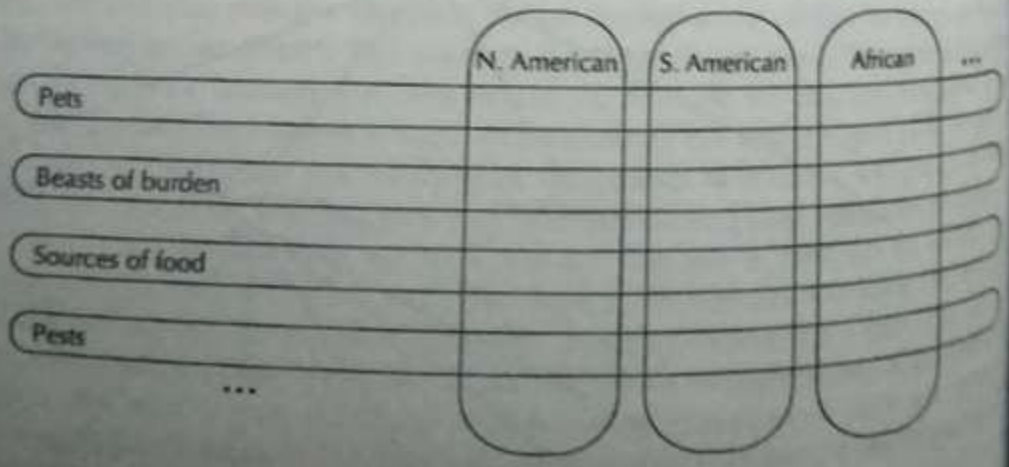


Figure 12.11 Example of Orthogonal Classification

EXERCISE 9-5: Define the functions 'name', 'age', 'salary', and 'hire-date' for accessing the parts of a personnel record; the functions 'firstn' and 'lastn' for accessing the parts of a name; and the functions 'month', 'day', and 'year' for accessing the parts of a date. Write expressions for accessing Don Smith's last name, salary, and the month in which he was hired.

Information Can Be Represented by Property Lists

A personnel record would probably not be represented in LISP in the way we have just described: It is too inflexible. Since each property of Don Smith is assigned a specific location in the list, it becomes difficult to change the properties associated with a person. A better arrangement is to precede each piece of information with an *indicator* identifying the property. For example,

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

This method of representing information is called a *property list* or *p-list*. Its general form is

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

in which each p_i is the indicator for a property and each v_i is the corresponding property value.

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

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

This flexibility is important in an experimental software environment in which all of the relevant properties may not be known at design time.

Information is selected from a simple list by various compositions of 'car' and 'cdr'. How can the properties of a *p*-list be accessed? We can attack this problem by considering how a person might solve it. Asked Don Smith's age, a person would probably begin searching from the left of the list for the indicator 'age'. The following element of the list is Don Smith's age. Let's consider this

process in more detail: Exactly how do we search a list from the left? We begin by looking at the first element of the list; if it's 'age', then the second element of the list is Don Smith's age, so we return it and we're done. If the first element of the list is not 'age', then we must skip the first two elements (the first property and its value) and repeat the process by checking the new first element.

Let's begin to express this in LISP notation. Suppose p is the property for which we're looking and x is the object which we're searching. We will write a 'getprop' function such that '(getprop p x)' is the value of the p property in property list x. First, we want to see if the first element of x is p; we can do this by '(eq (car x) p)'. If the first element of x is p, then the value of '(getprop p x)' is the second element of x, that is, '(cadr x)'. If the first element of x isn't p, then we want to skip over the first property of x and its value and continue looking for p. Notice that '(getprop p (cddr x))' will look for p beginning with the third element of x. We can summarize our algorithm as follows:

```
(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 $(c_1\ e_1)\ (c_2\ e_2) \cdots (c_n\ e_n)$)

The conditions $c_1, c_2, \ldots, c_n$ 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

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:

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

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,[7] 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)) )
```

7 Actually, an *a*-list is normally defined to be a list of *dotted* pairs. We will not address this detail until later.

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
```

Car and Cdr Access the Parts of Lists

We've described the kind of data values that lists are. We've seen in previous chapters, however, that there's much more to a data type than just data values. An *abstract data type* is a set of data values *together with* a set of operations on those data values. What are the primitive list-processing operations?

A complete set of operations for a composite data type, such as lists, requires operations for building the structures and operations for taking them apart. Operations that build a structure are called *constructors,*[4] and those that extract their parts are called *selectors.* LISP has one constructor—*cons*—and two selectors—*car* and *cdr.*

The first element of a list is selected by the 'car' function.[5] 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) )
```

Notice that, like 'car', 'cdr' requires a nonnull list for its argument (otherwise we can't remove its first element). Unlike 'car', 'cdr' *always* returns a list. This could be the null list; for example, '(cdr '(1))' returns '( )'.

It is important to realize that both 'car' and 'cdr' are *pure functions*, that is, they don't modify their argument list. The easiest way to think of the way they work is that they make a new copy of the list. For example, 'cdr' doesn't delete the first element of its argument; rather, it returns a new list exactly like its argument except without the first element. We will see later when we discuss the implementation of LISP lists that this copying does not actually have to be done.

'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) )
```

cdr ⇒ (30000 (August 25 1980))

cdr ⇒ ((August 25 1980))

car ⇒ (August 25 1980)

In general, the nth element of a list can be accessed by $n-1$ 'cdr's followed by a 'car'. Since '(car DS)' is this person's name '(Don Smith)', his first name is

```
(car (car DS))
  Don
```

and his last name is

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

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))

```
a => (Don Smith)
d => (Smith)
a => Smith
```

Arrays can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Numbers(1)	Numbers(2)	Numbers(3)	Numbers(4)	...

Arrays can be one- dimensional (like vectors), two-dimensional (like matrices) and Fortran allows you to create up to 7-dimensional arrays.

Declaring Arrays

Arrays are declared with the **dimension** attribute.

For example, to declare a one-dimensional array named number, of real numbers containing 5 elements, you write,

```
real, dimension(5) :: numbers
```

The individual elements of arrays are referenced by specifying their subscripts. The first element of an array has a subscript of one. The array numbers contains five real variables –numbers(1), numbers(2), numbers(3), numbers(4), and numbers(5).

To create a 5 x 5 two-dimensional array of integers named matrix, you write –

```
integer, dimension (5,5) :: matrix
```

You can also declare an array with some explicit lower bound, for example –

```
real, dimension(2:6) :: numbers
integer, dimension (-3:2,0:4) :: matrix
```

Assigning Values

You can either assign values to individual members, like,

```
numbers(1) = 2.0
```

or, you can use a loop,

```
do i  =1,5
   numbers(i) = i * 2.0
end do
```

One-dimensional array elements can be directly assigned values using a short hand symbol, called array constructor, like,

```
numbers = (/1.5, 3.2,4.5,0.9,7.2 /)
```

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.

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.

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.

Programming Paradigms

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.

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

*T*he 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.

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.

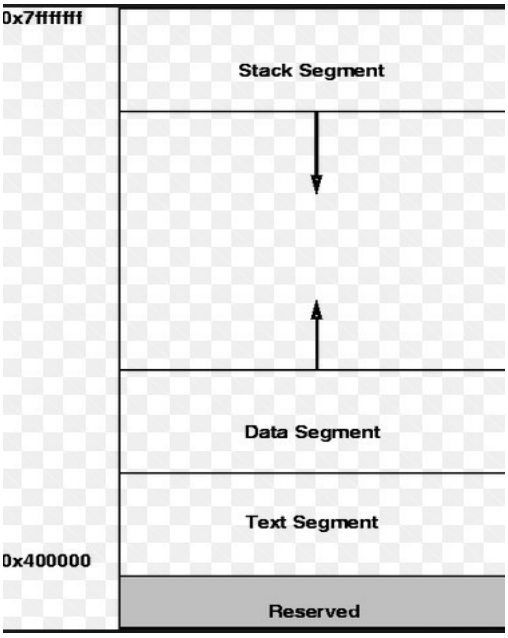
Some of these notations helped the programmer to design the:

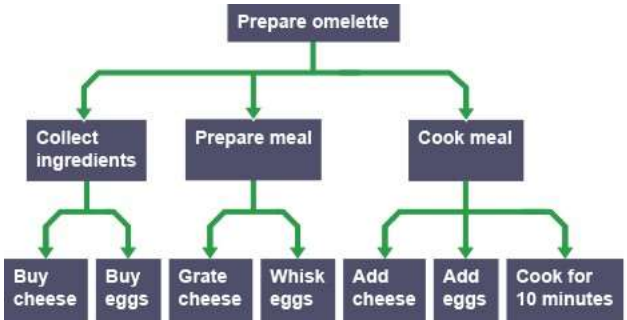
Pseudo-code

Control flow

- *Flow Diagrams*
- Later: Flowcharts

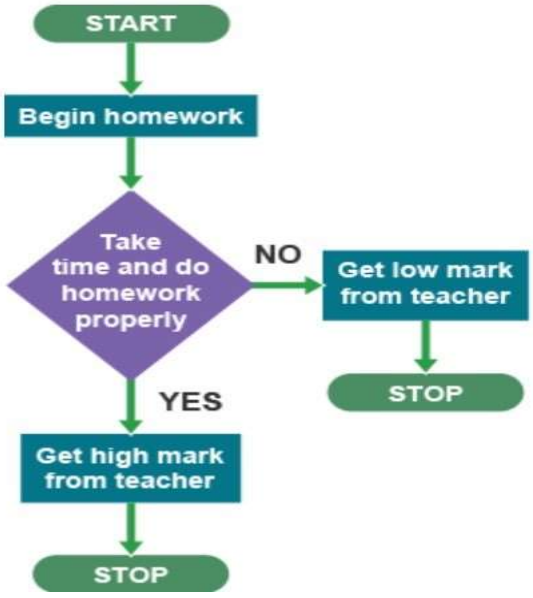
Memory layout





Flow charts

Flow charts show what is going on in a program and how data flows around it. Flow charts can represent everyday processes, show decisions taken and the result of these decisions.



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.

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 timeline

- 1954: Project approved
- 1957: FORTRAN
  - First version released
- 1958: FORTRAN II and III
  - Still many dependencies on IBM 704
- 1962: FORTRAN IV
  - "ANS FORTRAN" by American National Standards Institute
  - Breaks machine dependence
  - Few implementations follow the specifications
- We'll look at 1966 ANS FORTRAN

FORTRAN

- Goals
  - Decrease programming costs (to IBM)
  - Efficiency

Control Structure of FORTRAN

A control structure is a block of programming that analyses variables and chooses a direction in which to go based on given parameters. The term *flow control* details the direction the program takes (which way program control "flows"). Hence it is the basic decision-making process in computing; It is a prediction.

DESIGN: Control Structures

- Machine Dependence (1st generation)
- In FORTRAN, these were based on native IBM 704 branch instructions
  - "Assembly language for IBM 704"

FORTRAN II statement	IBM 704 branch operation
GOTO n	TRA k (transfer direct)
GOTO n, (n1, n2,…,nm)	TRA i (transfer indirect)
GOTO (n1, n2,…,nm), n	TRA i,k (transfer indexed)
IF (a) n1, n2, n3	CAS k
IF ACCUMULATOR OVERFLOW n1, n2	TOV k
…	…

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

n-way Branching with Computed GOTO

```
    GOTO (L1, L2, L3, L4 ), I
10  case 1
    GOTO 100
20  case 2
    GOTO 100
30  case 3
    GOTO 100
40  case 4
    GOTO 100
100
```

- Transfer control to label L_k if I contains k
- Jump Table

Loops

- Loops are implemented using combinations of IF and GOTOs
- Trailing-decision loop:

```
100 ...body of loop...
      IF (loop not done) GOTO 100
```

- Leading-decision loop:

```
100 IF (loop done) GOTO 200
      ...body of loop...
    GOTO 100
200 ...
```

- Readable?

But wait, there's more!

- Mid-decision loop:

```
100 ...first half of loop...
      IF (loop done) GOTO 200
      ...second half of loop...
    GOTO 100
200 ...
```

The DO-loop

- Fortunately, FORTRAN provides the DO-loop
- Higher-level than IF-GOTO-style control structures
  - No direct machine-equivalency

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

- I is called the *controlled variable*
- CONTINUE must have matching label
- DO allows stating what we *want*: higher level
  - Only built-in higher level structure

Nesting

- The DO-loop can be nested

```
DO 100 I = 1, N
     ...
     DO 200 J = 1, N
          ...
     200 CONTINUE
100 CONTINUE
```

  - They must be correctly nested
  - Optimized: controlled variable can be stored in index register
  - Note: we could have done this with GOTO

Name Structures

DESIGN: Name Structures

- What do name structures structure?
  - Names, of course!
- Primitives bind names to objects
  - INTEGER I, J, K
    - Allocate integers I, J, and K, and bind the names to memory locations
    - Declare: name, type, storage

Declarations

- Declarations are non-executable statements
- Unlike IF, GOTO, etc., which are executable statements
- Static allocation
  - Allocated once, cannot be deallocated for reuse
  - FORTRAN does not do dynamic allocation

Optional Declaration

- FORTRAN does not require variables to be declared
  - First use will declare a variable
- What's wrong with this?
  - COUNT = COUMT + 1
  - What if first use is not assignment?
- Convention:
  - Variables starting with letters i, j, k, l, m, n are integers
  - Others are floating point
  - Bad practice: Encourages funny names (KOUNT, ISUM, XLENGTH…)

Now: Semantics (meaning)

- "They went to the bank of the Rio Grande."
- What does this mean?
- How do we know?
- CONTEXT, CONTEXT, CONTEXT

Programming Languages

- X = COUNT(I)
- What does this mean
  - X integer or real
  - COUNT array or function
- Again Context
  - Set of variables visible when statement is seen
- Context is called ENVIRONMENT

SCOPE

- Scope of a binding of a name
  - Region of program where binding is visible
- In FORTRAN
  - Subprogram names GLOBAL
    - Can be called from anywhere
  - Variable names LOCAL
    - To subprogram where declared

Parameter Passing

Libraries

- Subprograms encourage libraries
  - Subprograms are independent of each other
  - Can be compiled separately
  - Can be reused later
  - Maintain library of already debugged and compiled useful subprograms

Parameter Passing

- Once we decide on subprograms, we need to figure out how to pass parameters
- Fortran parameters
  - Input
  - Output
    - Need address to write to
  - Both

Parameter Passing

- Pass by reference
  - On chance may need to write to
    - all vars passed by reference
  - Pass the address of the variable, not its value
  - Advantage:
    - Faster for larger (aggregate) data constructs
    - Allows output parameters
  - Disadvantage:
    - Address has to be de-referenced
      - Not by programmer—still, an additional operation
    - Values can be modified by subprogram
    - Need to pass size for data constructs - if wrong?

A Dangerous Side-Effect

- What if parameter passed in is not a variable?

```
SUBROUTINE SWITCH (N)
N = 3
RETURN
END
...
CALL SWITCH (2)
```

- The literal 2 can be changed to the literal 3 in FORTRAN's literal table!!!
  - I = 2 + 2 I = 6????
  - Violates security principle

Principles of Programming

- Security principle
  - No program that violates the definition of the language, or its own intended structure, should escape detection.

Pass by Value-Result

- Also called *copy-restore*
- Instead of pass by reference, copy the value of actual parameters into formal parameters
- Upon return, copy new values back to actuals
- Both operations done by caller
  - Can know not to copy meaningless result
    - E.g. actual was a constant or expression
- Callee never has access to caller's variables

FORFTRAN 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
```