Programming Paradigms

Module 1

Names, Scopes, and Bindings:-Names and Scopes, Binding Time, Scope Rules, Storage Management, Aliases, Overloading, Polymorphism, Binding of Referencing Environments. Control Flow: - Expression Evaluation, Structured and Unstructured Flow, Sequencing, Selection, Iteration, Recursion, Non-determinacy.

Introduction:

- A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer.

- They can be used to create programs that control the behavior of a machine and to express algorithms precisely.

The description of a programming language is usually split into the two components of **syntax (form) and semantics (meaning).**

1) Name in programming languages:
A name is a mnemonic character string used to represent something else.
- Names are a central feature of all programming languages.
- Earlier, numbers were used for all purposes, including machine addresses.
- Replacing numbers by symbolic names was one of the first major improvements to program notation.

e.g. variables, constants, executable code, data types, classes, etc.

In general, names are of two different types:
1. Special symbols: +, -, *
2. Identifiers**:** sequences of alphanumeric characters (in most cases beginning with a letter), plus in many cases a few special characters such as '_' or '$'.

Binding
A binding is an association between two things, such as a name and the thing it names
E.g. the binding of a class name to a class or a variables name to a variable.
Static and Dynamic binding: A binding is static if it first occurs before run time and remains unchanged throughout program execution.
A binding is dynamic if it first occurs during execution or can change during execution of the program.

2) Binding time
Binding time is the time when an association is established.
In programming, a name may have several attributes, and they may be bound at different times.
Example1:

<u>int *n*;</u>
n = 6;
first line binds the type int to *n* and the second line binds the value 6 to *n*. The first binding occurs when the program is compiled.
The second binding occurs when the program is executed.
<u>Examlpe2:</u>

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

In FORTRAN, addresses are bound to variable names at compile time.
The result is, in the compiled code, variables are addressed directly, without any indexing or other address calculations.
When the program is loaded, the address of the program is added to the address.
By the time execution begins, the "absolute" address of the variable is known.
FORTRAN is efficient, because absolute addressing is used.
It is inflexible, because all addresses are assigned at load time.
This leads to wasted space, and also prevents the use of direct or indirect recursion.
<u>Early and Late Binding:</u>
Early binding - efficiency
Late binding -flexibility
Different times at which decisions may be bound / Different types of binding times:
In the context of programming languages, there are few alternatives for binding time.

1. **Language design time**
2. **Language implementation time**
3. **Program writing time**
4. **Compile time**
5. **Link time**
6. **Load time**
7. **Run time**

<u>**1.** Language design time:</u>
The control flow constructs the set of fundamental types, the available constructors for creating complex types,etc are chosen when the language is designed.

During the design of the programming languages, the programmers decides what symbols should be used to represent operations
e.g. .Binding of operator symbols to operations

(* + …) (Multiplication, addition …)

2. <u>Language Implementation Time:</u>

Most language manuals leave a variety of issues to the language implementor.
Typical examples include the precision of the fundamental types, the coupling of I/O to the operating systems notion of files etc.
Bind data type, such as int in C to the range of possible values.
Example: the C language does not specify the range of values for the type int.

3. <u>Program writing time:</u>

Programmers choose algorithms, data structures and names.
While writing programs, programmers bind certain names with procedure, class etc.
Example: many names are bound to specific meanings when a person writes a program.

4. <u>Compile time:</u>

The time when a single compilation unit is compiled, while compiling the type of a variable can be identified.
Example: int c;   [at compile time int, c forms an association]

5. <u>Link time:</u> The time when all the compilation units comprising a single program are linked as the final step in building the program.

[The separate modules of a single program will be bound only at link time]

6. <u>Load time:</u>

Load time refers to the point at which the operating system loads the program into memory so that it can run.

7. <u>Run time:</u>

Run time is a very broad term that covers the entire span from the beginning to the end of execution. If we give the value of a variable during runtime, it is known as runtime binding

```
Ex.  Printf ("Enter the value of X");
Scanf ("%f", &X);
```

Scope: The textual region of the program in which a binding is active is its scope.

The scope of a name binding is the portion of the text of the source program in which that binding is in effect.
Using a name outside the scope of a particular binding implies one of two things: either it is undefined, or it refers to a different binding.

In C++, the scope of a local variable starts at the declaration of the variable and ends at the end of the block in which the declaration appears.
Scope is a **static** property of a name that is determined by the semantics of the PL and the text of the program.

Example:

```
class Foo
{         private int n;

          void foo() {
                  // 1
          }
```

```
            void bar() {
                        int m,n;
                        ...
// 2
            }
            ...
```

A reference to m at point 1 is undefined.
A reference to n at point 1 refers to an instance variable of Foo;

3)Scope rules

Difference between static and dynamic scoping: The scope rules (static, dynamic) of a language determine how references to names are associated with variables.

Static Scoping: Here the bindings between names and objects can be determined at compile time, by examining the text of the program.

Scope rules are more complex in FORTRAN.

- FORTRAN distinguishes between global and local variables.
- If a variable is not declared, it is assumed to be local to the current subroutine and to be of type integer if its name begins with the letters I-N, or real otherwise.
- Global variables in FORTRAN may be partitioned into common blocks, which are then imported by subroutines.

Nested scopes- Many programming languages allow scopes to be nested inside each other.
Example: Java actually allows classes to be defined inside classes or even inside methods, which permits multiple scopes to be nested.

```
    class Outer
            {
            int v1;  // 1

            void methodO()
            {
                    float v2; // 2

                    class Middle
                    {
                            char v3; // 3

                            void methodM()
                            {
                                    boolean v4; // 4

                                    class Inner
                                    {
                                            double v5; // 5
```

```
                                                void methodI()
                                                {
                                                        String v6; // 6
                                                }
                                        }
                                }
                        }
                }
                }
```

The scope of the binding for v1 the whole program
The scope of the binding for v2 is methodO and all of classes Middle and Inner, including their methods
The scope of the binding for v3 is all of classes Middle and Inner, including their methods
The scope of the binding for v4 is methodM and all of class Inner, including its method

```
void label_name (char *s) {
static short int n;
sprintf (s, "L%d\0", ++n);
/* C guarantees that static locals are initialized to zero */
/* "print" formatted output to s */
}
```

Fig: Code to illustrate the use of static variables
Consider the code in Figure.
The subroutine label_name can be used to generate a series of distinct character-string names: L1 , L2 , ...
A compiler might use these names in its assembly language output.

Nested Subroutines: The ability to nest subroutines inside each other, introduced in Algol 60, is a feature of many modern languages, including Pascal, Ada, ML, Python, Scheme, Common Lisp, and Fortran 90.

- Any constants, types, variables, or subroutines declared within a block are not visible outside that block in Algol-family languages.
- In Ada, a name may be prefixed by the name of the scope in which it is declared, using syntax that resembles the specification of fields in a record.
- My_proc.X , for example, refers to the declaration of X in subroutine My_proc , regardless of whether some other X has been declared in a lexically closer scope.

Access to Nonlocal Objects: The compiler can arrange for a frame pointer register to point to the frame of the currently executing subroutine at run time.
The simplest way in which to find the frames of surrounding scopes is to maintain a static link in each frame that points to the “parent” frame: the frame of the most recent invocation of the lexically surrounding subroutine.

- If a subroutine is nested k levels deep, then its frame’s static link, and those of its parent, grandparent, and so on, will form a static chain of length k at run time.
- To find a variable or parameter declared j subroutine scopes outward, target code at run time can dereference the static chain j times, and then add the appropriate offset.

Fig shows the Example of nested subroutines in Pascal. Vertical bars show the scope of each name.

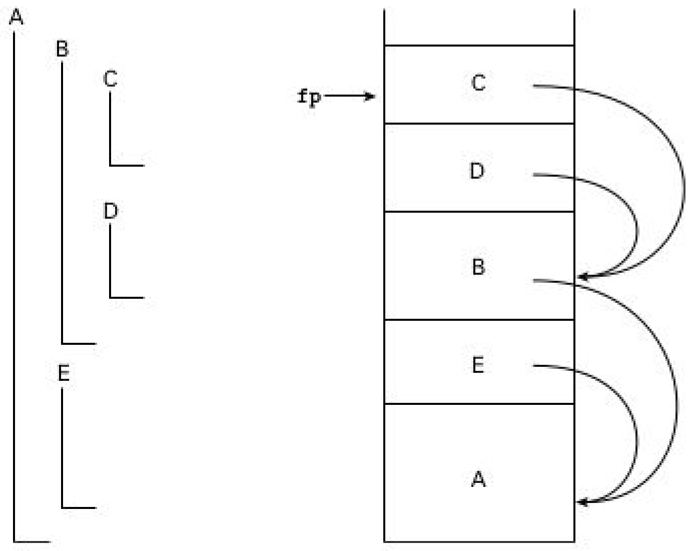
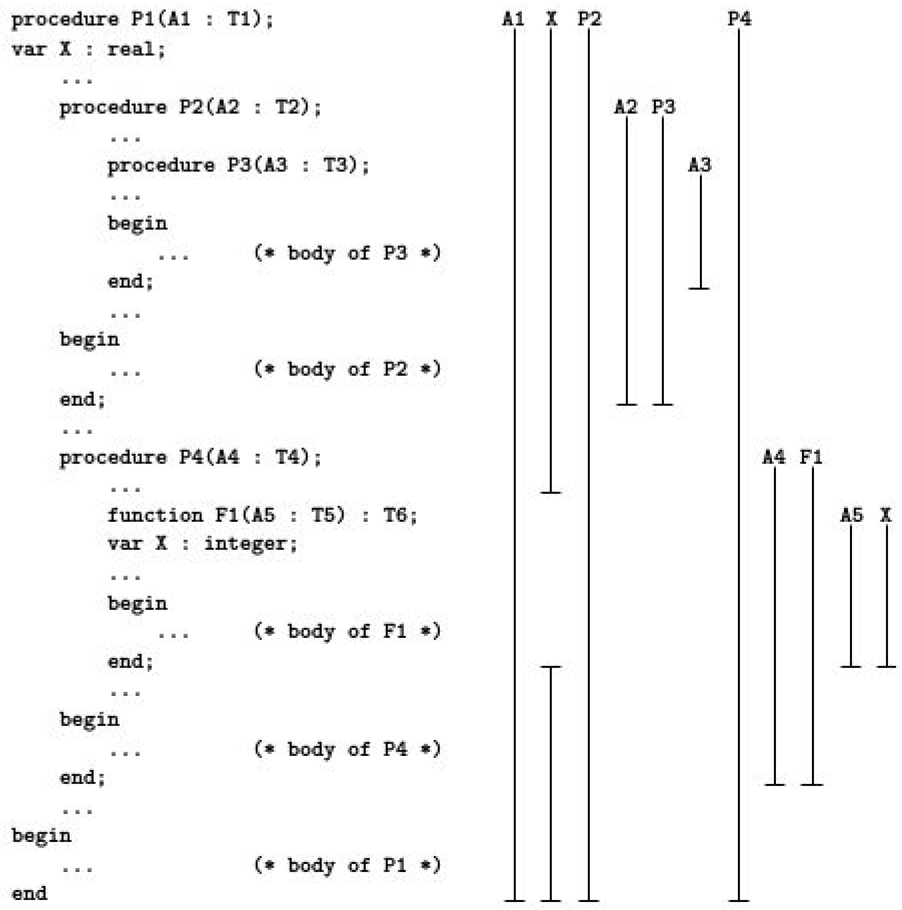


Fig shows Static chains.
Subroutines A , B , C , D , and E are nested as shown on the left.
If the sequence of nested calls at run time is A , E , B , D , and C , then the static links in the stack will look as shown on the right.
<u>Declaration order:</u> A field or method declared in a Java class can be used anywhere in the class, even before its declaration.
- A local variable declared in a method cannot be used before the point of its declaration.
Example:

```
class Demo
{
        public void method()
        {
                // Point 1

                int y;
        }

        private int x;
}
```



```
procedure P1(A1 : T1);
var X : real;
    ...
    procedure P2(A2 : T2);
        ...
        procedure P3(A3 : T3);
        ...
        begin
            ...     (* body of P3 *)
        end;
        ...
    begin
        ...         (* body of P2 *)
    end;
    ...
    procedure P4(A4 : T4);
        ...
        function F1(A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...     (* body of F1 *)
        end;
        ...
    begin
        ...         (* body of P4 *)
    end;
    ...
begin
    ...             (* body of P1 *)
end
```

The instance variable x can be used at Point 1, but not y

Example: C/C++:

```
void method2();        // Incomplete declaration

void method1();
{
        ...
        method2();
        ...
}

void method2()         // Definition completes the abov
{
        ...
        method1();
        ...
}
```

module

Mutual recursion

- Some Algol 60 compilers were known to process the declarations of a scope in program order.
- This strategy had the unfortunate effect of implicitly outlawing mutually recursive subroutines and types, something the language designers clearly did not intend.
- A Pascal compiler must scan the remainder of the scope's declarations to see if the name is hidden.
- To avoid complication, most Pascal successors specify that the scope of an identifier is the portion of that block from the declaration to the end.

The following is invalid in C#.

```
class A {
const int N = 10;
void foo() {
const int M = N; // uses inner N before it is declared
const int N = 20;
```

Declarations and Definitions: Recursive types and subroutines introduce a problem for languages that require names to be declared before they can be used.

How can two declarations each appear before the other?

C and C++ handle the problem by distinguishing between the declaration of an object and its definition.

Redeclarations: Some languages, particularly those that are intended for interactive use, permit the programmer to redeclare an object: to create a new binding for a given name in a given scope.

Interactive programmers commonly use redeclarations to fix bugs.

Modules: A major challenge in the construction of any large body of software is how to divide the effort among programmers in such a way that work can proceed on multiple fronts simultaneously.

- This modularization of effort depends critically on the notion of information hiding. It makes objects and algorithms invisible, whenever possible, to portions of the system that do not need them.

Encapsulating Data and Subroutines: The information hiding provided by nested subroutines is limited to objects whose lifetime is the same as that of the subroutine in which they are hidden. When control returns from a subroutine, its local variables will no longer be live.

Modules as Abstractions: A module allows a collection of objects—subroutines, variables, types, and so on—to be encapsulated in such a way that (1) objects inside are visible to each other, but (2) objects on the inside are not visible on the outside unless explicitly exported, and (3) objects outside are not visible on the inside unless explicitly imported.

Imports and Exports: Most module-based languages allow the programmer to specify that certain exported names are usable only in restricted ways.

This means that variables of that type may be declared, passed as arguments to the module's subroutines, and possibly compared or assigned to one another, but not manipulated in any other way.

Modules as Managers: Modules facilitate the construction of abstractions by allowing data to be made private to the subroutines that use them.

As in Figure, each module defines a single abstraction.

module

If we want to have several stacks, we must generally make the module a “manager” for instances of a stack type, which is then exported from the module, as shown in Figure.

Module Types and Classes: An alternative solution to the multiple instance problem can be found in Simula, Euclid, and ML, which treat modules as types, rather than simple encapsulation constructs.

Given a module type, the programmer can declare an arbitrary number of similar module objects.

Modules and separate compilation: To facilitate separate compilation, modules in many languages can be divided into a declaration part (header) and an implementation part (body), each of which occupies a separate file.

Code that uses the exports of a given module can be compiled as soon as the header exists; it is not dependent on the body.

Object Orientation: Many languages provide a class construct for object-oriented programming. Classes can be thought of as module types that have been augmented with an inheritance mechanism.

Module types and classes require only simple changes to the scope rules defined for modules.

Modules Containing Classes: There is a clear progression from modules to module types to classes, but it is not necessarily the case that classes are an adequate replacement for modules in all cases.

Class hierarchies may be just what we need to represent characters, possessions, buildings, goals, and a host of other data abstractions.

Dynamic Scoping

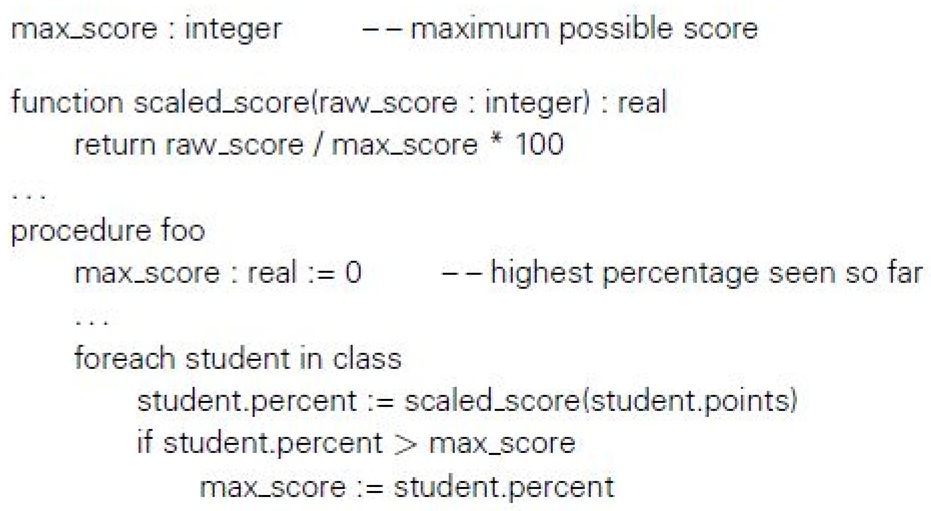
In a language with dynamic scoping, the bindings between names and objects depend on the flow of control at run time, and in particular on the order in which subroutines are called.

Dynamic scope rules are generally quite simple: the current binding for a given name is the one encountered most recently during execution, and not yet destroyed by returning from its scope.

The binding between names and objects in a language with dynamic scoping cannot in general be determined by a compiler.

```
Ex. Procedure Big is
        X:integer;
        Procedure sub1 is
          X:integer
          begin…….end
        Procedure sub2 is
         begin……….
                X;
              end;
         begin------
           end;
```

The use of dynamic scoping in Lisp and other early interpreted languages is that, it makes it very easy for an interpreter to look up the meaning of a name: all that is required is a stack of declarations.



```
max_score : integer         -- maximum possible score

function scaled_score(raw_score : integer) : real
    return raw_score / max_score * 100
. . .
procedure foo
    max_score : real := 0         -- highest percentage seen so far
    . . .
    foreach student in class
        student.percent := scaled_score(student.points)
        if student.percent > max_score
            max_score := student.percent
```

Fig: The problem with dynamic scoping.

Procedure scaled_score probably does not do what thmodulee programmer intended when dynamic scope rules allow procedure foo to change the meaning of max_score.

Local Scope:

- In block-structured languages, names declared modulein a block are local to the block.
- In Algol 60 and C++, local scopes can be as small as the programmer needs.
- Any statement context can be instantiated by a block that contains local variable declarations.

Local scope in C++

```
{
// x not accessible.
t x;
// x accessible
. . . .
{ // x still accessible in inner block
. . . .
}// x still accessible
}// x not accessible
```

Global Scope:

The name is visible throughout the program.

Global scope is useful for pervasive entities, such as library functions and fundamental constants (_ = 3.1415926) but is best avoided for application variables.

FORTRAN does not have global variables, although programmers simulate them by over using COMMON declarations.

Names declared at the beginning of the outermost block of Algol 60 and Pascal programs have global scope.

Implementing Scope

To keep track of the names in a statically scoped program, a compiler relies on a data abstraction called a *symbol table*.

The symbol table is a dictionary: it maps names to the information the compiler knows about them.
The most basic operations are to insert a new mapping or to look up the information that is already present for a given name.
Most variations on static scoping can be handled by augmenting a basic dictionary-style symbol table with enter scope and leave scope operations to keep track of visibility.
Object Lifetime:
The word "lifetime" is used in two slightly different ways

1. To refer to the lifetime of an object
2. The lifetime of the binding of a name to an object.

a. An object can exist before the binding of a particular name to it

```
Example (Java):
void something(Object o) {
        // 2
        }
        ....module
        Object p = new Object()
        // 1
        ....
        something(p);
        ....
        // 3

        (The object named by p exists at point 1, but the name o is not
        bound to it until point 2)
```

b. An object can exist after the binding of a particular name to it has ceased

```
Example:
void something(Object o) {
        // 2
        }
        ....
        Object p = new Object()module
        // 1
        ....
        something(p);
        ....
        // 3
module
```

c. A name can be bound to an object that does not yet exist

module

```
Example (Java)
Object o;
// 1
....
o = new Object(); // 2
```
(At point 1, the name o is bound to an object that does not come into existence until point 2)

d. A name can be bound to an object that has actually ceased to exist

```
Example (C++ - not possible in Java)
Object o = new Object();
...
delete o;          // 1
...
// 2
```
At 2, the name o is bound to an object that has ceased to exist.
(The technical name for this is a dangling reference).

4) Storage Allocation Mechanisms:

<u>Static (permanent) Allocation:</u>- The object exists during the entire time the program is running.

a. Global variables
- The precise way of declaring global variables differs from language to language
- In some languages (e.g. BASIC) any variable
- In FORTRAN any variable declared in the main program or in a COMMON block
- In Java, any class field explicitly declared static

b. Static local variables - only available in some languages

C/C++ local variables explicitly declared static

```
int foo() {
                    static int i;
                    ...
```

c. Many constants
Advantages: efficiency (direct addressing),history-sensitive subprogram support.
Disadvantage: lack of flexibility

Along with local variables and elaboration-time constants, the compiler typically stores a variety of other information associated with the subroutine, including:

Arguments and return values. Modern compilers keep these in registers whenever possible, but sometimes space in memory is needed.

Temporaries. These are usually intermediate values produced in complex calculations.
A good compiler will keep them in registers whenever possible.

Bookkeeping information. This may include the subroutine's return address, a reference to the stack frame of the caller (also called the *dynamic link*), additional saved registers, debugging information, and various other values.

Stack Based Allocation:-Storage bindings are created for variables when their declaration statements are elaborated.

Typically, the local variables and parameters of a method have stack lifetime.
This name comes from the normal way of implementing routines, regardless of language.

Since routines obey a LIFO call return discipline, they can be managed by using a stack composed of stack frames - one for each currently active routine.

Example:

```
void d() { /* 1 */ }
void c() { ... d() ... }
void b() { ... c() ... }
void a() { ... b() ... }
int main() { ... a() ... }
```

Stack at point 1:

```
-------------------
| Frame for d |          <-- top of stack
-------------------
| Frame for c |
-------------------
| Frame for b |
-------------------
| Frame for a |
-------------------
| Frame for main |
-------------------
```

- In a language without recursion, it can be advantageous to use a stack for local variables, rather than allocating them statically.

- As a result, the total space needed for local variables of currently active subroutines is seldom as large as the total space across all subroutines, active or not.
- A stack may therefore require substantially less memory at run time than would be required for static allocation.

module

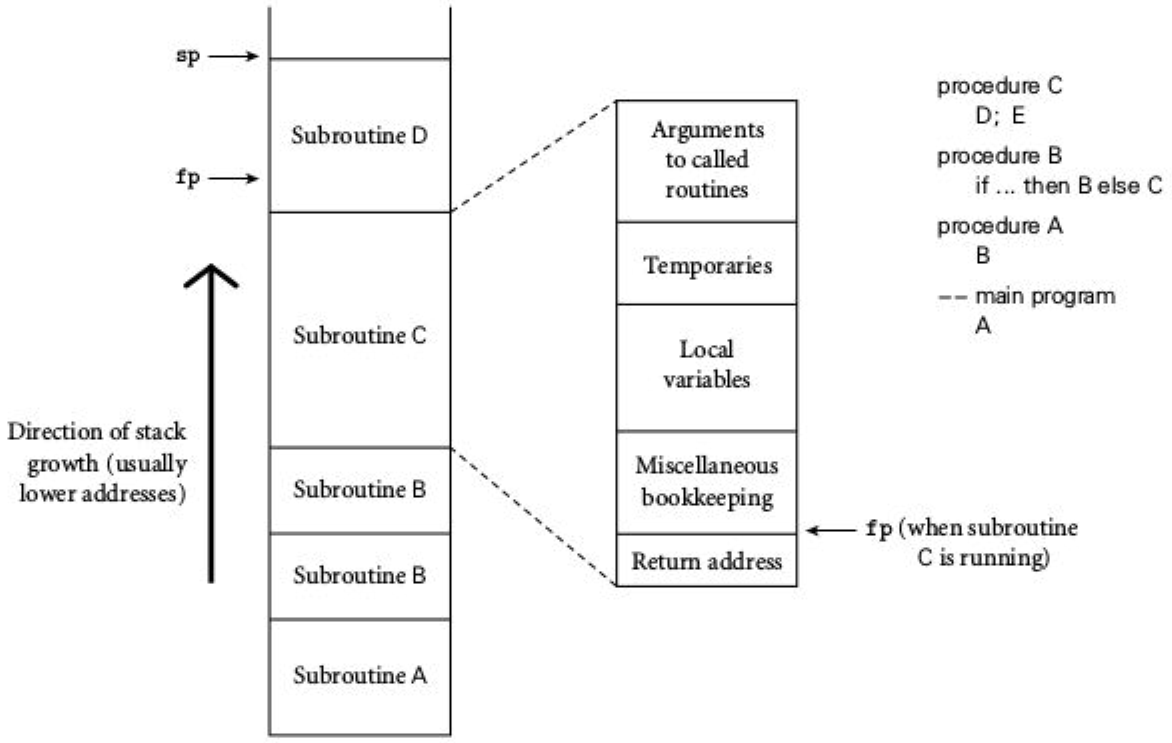


Fig: Stack-based allocation of space for subroutines.
Assume here that subroutines have been called as shown in the upper right.
In particular, B has called itself once, recursively, before calling C .
If D returns and C calls E , E ’s frame (activation record) will occupy the same space previously used for D ’s frame.
The stack pointer (sp) register points to the first unused location on the stack and the frame pointer (fp) register points to a known location within the frame of the current subroutine.
Advantage: allows recursion; conserves storage
Disadvantages: Overhead of allocation and deallocation
Subprograms cannot be history sensitive
Inefficient references (indirect addressing)

__**Heap Based Allocation**__ –A heap is a region of storage in which subblocks can be allocated and deallocated at arbitrary times.
Heaps are required for the dynamically allocated pieces of linked data structures, and for object like fully general character strings, lists etc. whose size may change as a result of an assignment statement or other update operation.

The shaded blocks are in use, the clear blocks are free.
Cross hatched space at the ends of in use blocks represents internal fragmentation.
The discontiguous free blocks indicate external fragmentation.

- Internal fragmentation occurs when a storage management algorithm allocates a block that is larger than required to hold a given object, the extra space is then unused.
- External fragmentation occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks.
- As the program runs, Heap space grows as objects are created.
- A language implementation typically uses one of three approaches to "recycling" space used by objects that are no longer alive:

  Explicit - the program is responsible for releasing space needed by objects that are no longer needed using some construct such as delete (C++).

  Reference counting: each heap object maintains a count of the number of external pointers/references to it.
  When this count drops to zero, the space utilized by the object can be reallocated

Garbage collection: The process of deallocating the memory given to a variable is known as garbage collection.
The system can work in proper order only by proper garbage collection.
C, C++ does not do garbage collection implicitly but java has implicit garbage collector.

Advantage: Provides for dynamic storage management.

Disadvantage: Inefficient (instead of static) and unreliable

- The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program.
- Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*.
- Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming.
  Deallocation errors are difficult to identify and fix.

- Both language designers and programmers consider automatic garbage collection an essential language feature.

5) The Meaning of Names within a Scope

- Assume that, there is a one-to-one mapping between names and visible objects in any given point in a program.

- A name that can refer to more than one object at a given point in the program is said to be overloaded.

Aliasing: Two or more names that refer to the same object at the same point in the program are said to be aliases.

1. Aliases can be created by assignment of pointers/references

```
Example: Java:          Robot karel = ...
                             Robot foo = karel;

                             foo and karel are aliases for the same Robot object
```

2. Aliases can be created by passing reference parameters.
Example: C++

```
void something(int a [], int & b)
{
        // 1
        ...
}

int x [100];
int y;
something(x, y);
```

After the call to something(x, y), at point 1 x and a are aliases for the same array.

```
double sum, sum_of_squares;
...
void accumulate(double& x)      // x is passed by reference
{
    sum += x;
    sum_of_squares += x * x;
}
...
accumulate(sum);
```

- If sum is passed as an argument to accumulate , then sum and x will be aliases for one another, and the program will probably not do what the programmer intended.
- Given import lists, the compiler can identify when a subroutine call would create an alias, and the language can prohibit it.

- Aliases tend to make programs more confusing than they otherwise would be.

6) Overloading:
A name is said to be overloaded if, in some scope, it has two or more meanings, with the actual meaning being determined by how it is used.

Example: C++
```
void something(char x)
...
void something(double x)
...
// 1
```

At point 1, something can refer to one or the other of the two methods, depending on its parameter.

- Within the symbol table of a compiler, overloading must be handled by arranging for the lookup routine to return a list of possible meanings for the requested name.
- The semantic analyzer must choose from among the elements of the list based on context.
- Most languages that allow overloaded enumeration constants allow the programmer to provide appropriate context explicitly.
- In Ada, for example, one can say

```
print(month'(oct));
```

In Modula-3 and C#, every use of an enumeration constant must be prefixed with a type name, even when there is no chance of ambiguity:
```
mo := month.dec;
pb := print_base.oct;
```
Redefining Built-in Operators: Ada, C++, C#, Fortran 90, and Haskell also allow the built-in arithmetic operators (+ , - , * , etc.) to be overloaded with user-defined functions.
In C++ and C#, which are object-oriented, A + B may be short for either operator+(A, B) or A.operator+(B) .
In the latter case, A is an instance of a class (module type) that defines an operator+ function.
In C++:
```
class complex {
double real, imaginary;
...
public:
complex operator+(complex other) {
return complex(real + other.real, imaginary + other.imaginary);
}
...
};
...
complex A, B, C;
...
C = A + B;                              // uses user-defined operator+
```

C# syntax is similar.

7) Polymorphism:

Polymorphism is the concept that supports the capability of an object of a class to behave differently in response to a message or action.

1. Compile-time (static) polymorphism: the meaning of a name is determined at compile-time from the declared types of what it uses

2. Run-time (dynamic) polymorphism: the meaning of a name is determined when the program is running from the actual types of what it uses.

Example: Java has both types in different contexts:
a. When a name is overloaded in a single class, static polymorphism is used to determine which declaration is meant.
b. When a name is overridden in a subclass, dynamic polymorphism is used to determine which version to use.

Eg: Suppose we wish to compute the minimum of two values of either integer or floating-point type. In Ada we might obtain this capability using overloaded functions:
function min(a, b : integer) return integer is ...
function min(x, y : real) return real is ...
In C, however, we could get by with a single function:
double min(double x, double y) { ...

- If the C function is called in a context that expects an integer (e.g., i = min(j,k)), the compiler will automatically convert the integer arguments (j and k) to floating-point numbers, call min , and then convert the result back to an integer (via truncation).
- Coercion is the process by which a compiler automatically converts a value of one type into a value of another type when that second type is required by the surrounding context.
- Ada coerces nothing but explicit constants, subranges, and in certain cases arrays with the same type of elements.
- C will perform same coercions on arguments to functions.
- Coercion and overloading can have very different costs.

With the implicit parametric polymorphism of Lisp, ML, and their descendants, the programmer need not specify a type parameter.
The Scheme definition of min looks like this:
(define min (lambda (a b) (if (< a b) a b)))
The typical Scheme implementation employs an interpreter that examines the arguments to min and determines, at run time, whether they support a < operator.
6) The Binding of Referencing Environments
Scope rules determine the referencing environment of a given statement in a program.

Static scope rules specify that the referencing environment depends on the lexical nesting of program blocks in which names are declared.
Dynamic scope rules specify that the referencing environment depends on the order in which declarations are encountered at run time.

```
type person = record
      . . .
      age : integer
      . . .
threshold : integer
people : database

function older_than_threshold(p : person) : boolean
      return p.age ≥ threshold

procedure print_person(p : person)
      -- Call appropriate I/O routines to print record on standard output.
      -- Make use of nonlocal variable line_length to format data in columns.
      . . .

procedure print_selected_records(db : database;
            predicate, print_routine : procedure)
      line_length : integer

      if device_type(stdout) = terminal
            line_length := 80
      else        -- Standard output is a file or printer.
            line_length := 132
      foreach record r in db
            -- Iterating over these may actually be
            -- a lot more complicated than a 'for' loop.
            if predicate(r)
                  print_routine(r)

-- main program
. . .
threshold := 35
print_selected_records(people, older_than_threshold, print_person)
```

Figure : Program to illustrate the importance of binding rules.
A dynamic scoping example appears in Figure.
Procedure print selected_records is assumed to be a general-purpose routine that knows how to traverse the records in a database, regardless of whether they represent people, sprockets, or salads.
It takes as parameters a database, a predicate to make print/don't print decisions, and a subroutine that knows how to format the data in the records of this particular database.

- In a language with dynamic scoping, it is natural for procedure print selected records to declare and initialize this variable locally, knowing that code inside print_routine will pick it up if needed.
- The late binding of the referencing environment of a subroutine that has been passed as a parameter is known as shallow binding.

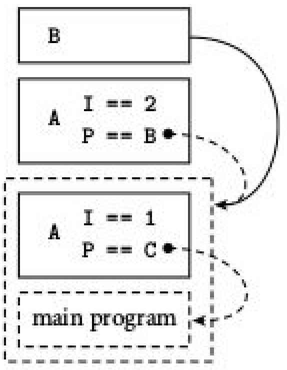
Subroutine Closures: Deep binding is implemented by creating an explicit representation of a referencing environment and bundling it together with a reference to the subroutine.

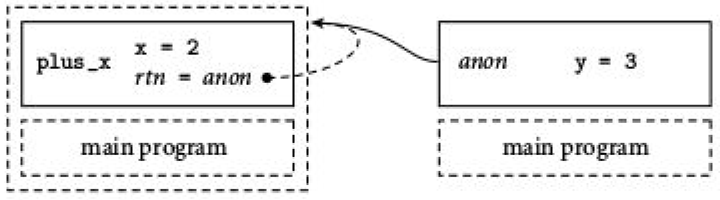
- The bundle as a whole is referred to as a closure.

- Usually the subroutine itself can be represented in the closure by a pointer to its code.

First-Class Values and Unlimited Extent: A value in a programming language is said to have first-class status if it can be passed as a parameter, returned from a subroutine, or assigned into a variable.
Simple types such as integers and characters are first-class values in most programming languages.
A "second-class" value can be passed as a parameter, but not returned from a subroutine or assigned into a variable, and a "third-class" value cannot even be passed as a parameter.
Fig shows the need for unlimited extent.
When function plus_x is called in Example, it returns (left side of the fig:) a closure containing an anonymous function.

```
program binding_example(input, output);

procedure A(I : integer; procedure P);

    procedure B;
    begin
        writeln(I);
    end;

begin (* A *)
    if I > 1 then
        P
    else
        A(2, B);
end;

procedure C; begin end;

begin (* main *)
    A(1, C);
end.
```





When the anonymous function is subsequently called (right side of the figure), it must be able to access variables in the closure's environment—in particular, the x inside plus_x .

Object Closures: The referencing environment in a closure will be nontrivial only when passing a nested subroutine.
This means that the implementation of first-class subroutines is trivial in a language without nested subroutines.

An object that plays the role of a function and its referencing environment may variously be called an object closure, a function object, or a functor.
Object closures are sufficiently important that some languages support them with special syntax. In C++, an object of a class that overrides operator() can be called as if it were a function:

```
class int_func {
public:
virtual int operator()(int i) = 0;
};
class plus_x : public int_func {
const int x;
public:
plus_x(int n) : x(n) { }
virtual int operator()(int i) { return i + x; }
};
...
plus_x f(2);
cout << f(3) << "\n";                // prints 5
```

Object f could also be passed to any function that expected a parameter of class int_func .

The compiler arranges to allocate it in the heap, and to refer to it indirectly through a hidden pointer, included in the closure.
C# 3.0 provides an alternative lambda expression notation for anonymous methods.
The (one-line) body of PlusY above could be replaced with,
return i => i + y;

7) Control flow:
The order in which operations are executed in a program
e.g. in C++ like language,

```
 a = 1;
b = a + 1;
 if a > 100 then b = a - 1; else b = a + 1;
 a - b + c
```

Eight major categories of control flow mechanisms:

a. Sequencing: - Statements are to be executed in a certain specified order- usually the order in which they appear in the program text.
b. Selection: - Depending on some run time condition, a choice is to be made among two or more statements or expressions.
   If and case statements are referred to as alternation.
c. Iteration: - A given fragment of code is to be executed repeatedly, either a certain number of times, or until a certain run- time condition is true.
   Iteration constructs include for/do, while, and repeat loops.
d. Procedural abstraction: - A potentially complex collection of control constructs is encapsulated in a way that allows it to be treated as a single unit, usually subject to parameterization.

e. Recursion: - An expression is defined in terms of itself, either directly or indirectly; the computational model requires a stack on which to save information about partially evaluated instances of the expression.
   Recursion is usually defined by means of self-referential subroutines.
f. Concurrency:- Two or more program fragments are to be executed/evaluated at the same time, either in parallel on separate processors, or interleaved on a single processor in a way that achieves the same effect.
g. Exception handling and speculation:- A program fragment is executed optimistically, on the assumption that some expected condition will be true.
   If that condition turns out to be false, execution branches to a handler that executes in place of the remainder of the protected fragment or in place of the entire protected fragment.
h. Nondeterminacy: - The ordering or choice among statements or expressions is deliberately left unspecified, implying that any alternative will lead to correct results.

Operators from other sort of functions:
An expression generally consists of either a simple object or an operator or function applied to a collection of operands or arguments, each of which in turn is an expression.
Function call consists of a function name followed by a parenthesized, comma-separated list of arguments, as in my_func (A, B, C)

Operators are typically simpler, taking only one or two arguments, and dispensing with the parentheses and commas:

```
a + b
 -c
```

A language may specify that function calls employ prefix, infix, or postfix notation.
These terms indicate, respectively, whether the function name appears before, among, or after its several arguments:

```
Prefix: op a b
Infix: a op b
Postfix: a b op
```

Prefix, infix, and postfix notation. / Cambridge polish notation:
Most imperative languages use infix notation for binary operators and prefix notation for unary operators and other functions.
Lisp uses prefix notation for all notation for all functions.
Cambridge polish notation places the function name inside the parentheses:

```
( * ( + 1 3) 2)
(append a b c my_list)
```

L- value, r-value:

Consider the following assignments in C:

```
d= a;
a= b+c;
```

In the first statement, the right- hand side of the assignment refers to the value of a, which we wish to place into d.

In the second statement, the left hand side refers to the location of a, where we want to put the sum of b and c.
Because of the use on the left hand side of assignment statements, expressions that denote locations are referred to as L-values.
Expressions that denote values are referred to as r- values.
A given expression can be either an L-value or an r-value, depending on the context in which it appears.
It makes no sense to say 2+3 =a, or even a= 2+3, if a is the name of a constant.
In C one may write

```
(f (a) +3) -> b [c]=2;
```

In this expression f (a) returns a pointer to some element of an array of pointers to structures. The assignment places the value 2 into the c-th element of field b of the structure. **Orthogonality in the context of programming language design:** One of the principal design goals of Algol 68 was to make the various features of the languages as orthogonal as possible.
Orthogonality means that features can be used in any combination, the combinations all make sense, and the meaning of a given feature is consistent, regardless of the other features with which it is combined.
Algol 68 was one of the first languages to make orthogonality a principal design goal.
The following, for example is valid in Algol 68:

```
begin
  a:=if b < c then d else e;
  a:= begin f(b); g(c) end;
  g(d);
  2+3
End
```

8) Expression Evaluation- An expression consists of:

- o A simple object, e.g. number or variable
- o An operator applied to a collection of operands or arguments which are expressions

- Common syntactic forms for operators:
  - o Infix notation for binary operators, e.g. A + B
  - o Prefix notation for unary operators, e.g. -A
  - o Cambridge Polish notation, e.g. (* (+ 1 3) 2) in Lisp =(1+3)*2=8 etc..
- Smalltalk uses infix notation for all functions (which it calls messages), both built-in and user-defined.

- The following Smalltalk statement sends a “ displayOn: at: ” message to graphical object myBox , with arguments myScreen and 100@50 (a pixel location).

- It corresponds to what other languages would call the invocation of the “ displayOn: at: ” function with arguments myBox , myScreen , and 100@50 .

- myBox displayOn: myScreen at: 100@50

- In Algol one can say a := if b <> 0 then a/b else 0;

- Here “ if . . . then . . . else ” is a three-operand infix operator. The equivalent operator in C is written “. . . ? . . . : . . . ”:

a = b != 0 ? a/b : 0;

- Postfix notation is used for most functions in Postscript, Forth, the input language of certain hand-held calculators, and the intermediate code of some compilers.

Expression Evaluation Ordering: Precedence and Associativity

Precedence rules specify that certain operators, in the absence of parentheses, group more tightly than other operators.
In Java all binary operators except assignments are left associative
3 - 3 + 5
x = y = f()
In C++ arithmetic operators (+, -, *, ...) have higher precedence than relational operators (<, ...)
x + y < z + w
x + y == z

The use of infix, prefix, and postfix notation leads to ambiguity as to what is an operand of what, e.g. a+b*c**d**e/f in Fortran.

The choice among alternative evaluation orders depends on:

Operator precedence: higher operator precedence means that a (collection of) operator(s) group more tightly in an expression than operators of lower precedence

left associative: operators are evaluatated left-to-right (most common)

right associative: operators are evaluated right-to-left   etc..

Evaluation Order in Expressions:

- Precedence and associativity define rules for structuring expressions

- Expression a-f(b)-c*d is structured as (a-f(b))-(c*d) by compiler, but either (a-f(b)) or (c*d) can be evaluated first at run-time

- *Side effects:* e.g. if f(b) above modifies d (i.e. f(b) has a side effect) the expression value will depend on the operand evaluation order

- Code improvement: compilers rearrange expressions to maximize efficiency

  - Improve memory loads:
    a:=B[i];   load a from memory
    c:=2*a+3*d; compute 3*d first, because waiting for a to arrive in processor

  - Common subexpression elimination:
    a:=b+c;
    d:=c+e+b;   rearranged as d:=b+c+e, it can be rewritten into d:=a+e

Expression Reordering Problems

- Rearranging expressions may lead to arithmetic overflow or different floating point results

  - Assume b, d, and c are very large positive integers, then if b-c+d is rearranged into (b+d)-c arithmetic overflow occurs

  - Floating point value of b-c+d may differ from b+d-c

  - Most programming languages will not rearrange expressions when parenthesis are used, e.g. write (b-c)+d to avoid problems

- Java: expressions evaluation is always left to right and overflow is always detected.

- Short-Circuit Evaluation: *Short-circuit evaluation* of Boolean expressions means that computations are skipped when logical result of a Boolean operator can be determined from the evaluation of one operand.

  - C, C++, and Java use c*onditional and/or* operators: && and ||

    - If a in a&&b evaluates to false, b is not evaluated

    - If a in a||b evaluates ot true, b is not evaluated

    - Avoids the Pascal problem

- Pascal does not use short-circuit evaluation
    - The program fragment below has the problem that element a[11] can be accessed resulting in a dynamic semantic error:
    - **var** a:**array** [1..10] **ofinteger**;
      ```
      ...
      i:=1;
      while i<=10 and a[i]<>0 do
        i:=i+1
      ```

<u>Assignments:</u>

- Assignment a:=b
    - Left-hand side a of the assignment is a location, called *l-value* which is an expression that should denote a location
    - Right-hand side b of the assignment is a value, called *r-value* which is an expression.
    - Languages that adopt *value model* of variables copy values: Ada, Pascal, C, C++ copy the value of b into the location of a
    - Languages that adopt *reference model* of variables copy references: Clu copies the reference of b into a and both a and b refer to the same object
- Variable initialization
    - *Implicit*: e.g. 0 or **NaN** (not a number) is assigned by default
    - *Explicit*: by programmer (more efficient than using an explicit assignment, e.g. **int** i=1; declares i and initializes it to 1 in C)
- Zero-initialization applies recursively to the sub-components of variables of user-defined composite types.
- Combination of assignment operators
    - In C/C++ a+=b is equivalent to a=a+b (but a[i++]+=b is different from a[i++]=a[i++]+b)
    - Compiler produces better code, because the address of a variable is only calculated once

● C provides prefix and postfix increment and decrement operations.
A[index_fn(i)]++;

Or ++A[index_fn(i)];

Increment and decrement operators provide elegant syntax for code that uses an index or a pointer to traverse an array:
A[--i] = b;
*p++ = *q++;

Multiway assignments in Clu, ML, and Perl

- o a,b := c,d assigns c to a and d to b *simultaneously*, e.g. a,b := b,a swaps a with ba,b := 1 assigns 1 to both a and b

Multiway assignment allows functions to return tuples, as well as single values:

a, b, c = foo(d, e, f);
References and Values : Assignment appears to be a very straightforward operation.
There are some subtle but important differences in the semantics of assignment in different imperative languages.
Consider the following assignments in C:

d = a;
a = b + c;
In the first statement, the right-hand side of the assignment refers to the value of a , which we wish to place into d.
In the second statement, the left-hand side refers to the location of a , where we want to put the sum of b and c .
Dynamic Checks: Instead of giving every uninitialized variable a default value, a language or implementation can choose to define the use of an uninitialized variable as a dynamic semantic error, and can catch these errors at run time.
Definite Assignment: For local variables of methods, Java and C# define a notion of definite assignment that precludes the use of uninitialized variables.
This notion is based on the control flow of the program, and can be statically checked by the compiler.

Boxing: A drawback of using a value model for built-in types is that they can't be passed uniformly to methods that expect class-typed parameters.

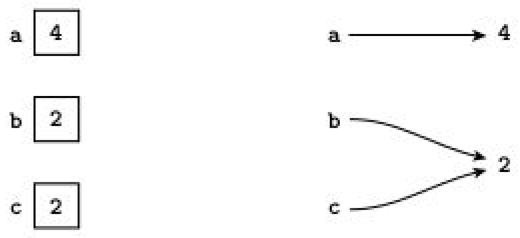


Fig: The value (left) and reference (right) models of variables.

Early versions of Java required the programmer to “wrap” objects of built-in types inside corresponding predefined class types in order to insert them in standard container (collection) classes:

```
import java.util.Hashtable;
...
Hashtable ht = new Hashtable();
...
Integer N = new Integer(13);          // Integer is a "wrapper" class
ht.put(N, new Integer(31));
Integer M = (Integer) ht.get(N);
int m = M.intValue();
```

<u>Constructors:</u> Many object-oriented languages allow the programmer to define types for which initialization of dynamically allocated variables occurs automatically, even when no initial value is specified in the declaration.
Initialization is interpreted as a call to a constructor function for the variable’s type, with the initial value as an argument.

<u>Ordering within Expressions</u>

Precedence and associativity rules define the order in which binary infix operators are applied within an expression.
In the expression,
a - f(b) - c * d
From associativity f(b) will be subtracted from a before performing the second subtraction.
From precedence the right operand of that second subtraction will be the result of c * d, but we do not know whether a - f(b) will be evaluated before or after c * d.
There are two main reasons why the order can be important:
1. *Side effects:* If f(b) may modify d, then the value of a - f(b) - c * d will depend on whether the first subtraction or the multiplication is performed first.
2. *Code improvement:* The order of evaluation of sub expressions has an impact on both register allocation and instruction scheduling.
In the expression a *b + f(c), it is probably desirable to call f before evaluating a * b, because the product, if calculated first, would need to be saved during the call to f.
<u>Applying Mathematical Identities</u>

Some language implementations allow the compiler to *rearrange* expressions involving operators whose mathematical abstractions are commutative, associative, and/or distributive, in order to generate faster code.
Consider the following Fortran fragment:
a = b + c
d = c + e + b
Some compilers can recognize the *common subexpression* in the first and second statements, and generate code equivalent to
a = b + c
d = a + e
Similarly,
a = b/c/d
e = f/d/c
may be rearranged as
t = c * d
a = b/t
e = f/t

Short circuit Boolean evaluation/ Use:

Short-circuit evaluation of Boolean expressions means that computations are skipped when logical result of a Boolean operator can be determined from the evaluation of one operand.

C, C++, and Java use c*onditional and/or* operators: && and ||

a. If a in a&&b evaluates to false, b is not evaluated

b. If a in a||b evaluates ot true, b is not evaluated

c. Useful to increase program efficiency, e.g.
   if (unlikely_condition && expensive_condition()) ...

Ada conditional and/or uses **then** keyword, e.g.: cond1 **andthen** cond2

Ada, C, and C++ also have regular Boolean operators

- Short circuiting is not necessarily as attractive for situations in which a Boolean subexpression can cause a side effect.
- A few languages provide both regular *and* short-circuit Boolean operators.

found_it := p /= null and then p.key = val;
...
if d = 0 or else n/d < threshold then ...
(Ada uses /= for "not equal.")

Structured and Unstructured Flow

Unstructured flow: the use of **goto** statements and *statement labels* to obtain control flow .

Useful for jumping out of nested loops and for programming errors and exceptions

Java has no **goto** statement

Structured flow*:*

- Sequencing: the subsequent execution of a list of statements in that order
- Selection: **if-then-else** statements and **switch** or **case**-statements
- Iteration: **for** and **while** loop statements
- Subroutine calls and recursion

Structured Alternatives to goto
Most of the controversy surrounding gotos revolved around a small number of special cases, each of which was eventually addressed in structured ways.
Where once a goto might have been used to jump to the end of the current subroutine, most modern languages provide an explicit return statement.
Where once a goto might have been used to escape from the middle of a loop, most modern languages provide a break or exit statement for this purpose.
Multilevel Returns
Returns and (local) gotos allow control to return from the current subroutine. On occasion it may make sense to return froma *surrounding* routine.
The search routine might invoke several nested routines, or a single routine multiple times, once for each place in which to search.
In such a situation Algol 60, PL/I, and Pascal, permit a goto to branch to a lexically visible label *outside* the current subroutine.
Errors and Other Exceptions
A deeply nested block or subroutine may discover that it is unable to proceed with its usual function, and moreover lacks the contextual information it would need to recover in any graceful way.
Eiffel formalizes this by saying that every software component has a *contract*—a specification of the function it performs.
A component that is unable to fulfill its contract is said to *fail*. It must arrange for control to "back out" to some context in which the program is able to recover.
Conditions that require a program to "back out" are usually called *exceptions.*
Continuations: The notion of nonlocal gotos that unwind the stack can be generalized by defining what are known as *continuations*.
A continuation consists of a code address and a referencing environment to be restored when jumping to that address.
In higher-level terms, a continuation is an abstraction that captures a *context* in which execution might continue.

- Continuation support in Scheme takes the form of a general-purpose function called call-with-current-continuation, sometimes abbreviated call/cc.

- This function takes a single argument *f* , which is itself a function. It calls *f* , passing as argument a continuation *c* that captures the current program counter and referencing environment.
- **Sequencing**

One statement appearing after another.

- A list of statements in a program text is executed in top-down order
- A *compound statement* is a delimited list of statements

  - A compund statement is a *block* when it includes variable declarations
  - C, C++, and Java use { and } to delimit a block
  - Pascal and Modula use **begin** ... **end**
  - Ada uses **declare** ... **begin** ... **end**

C, C++, and Java: expressions can be used where statements can appear.

In pure functional languages, sequencing is impossible.

- In most imperative languages, lists of statements can be enclosed with begin. . . end or {. . . } delimiters and then used in any context in which a single statement is expected. Such a delimited list is usually called a *compound statement*.
- A compound statement optionally preceded by a set of declarations is sometimes called a *block*.

The various sequencing constructs in Lisp are used only in program fragments that do not conform to a purely functional programming model.

- **Selection**

Selects which statements to execute next.
Forms of **if-then-else** selection statements:

  - C and C++ EBNF syntax:
    if (<expr>) <stmt> [**else**<stmt>]

Condition is integer-valued expression.

When it evaluates to 0, the *else-clause* statement is executed otherwise the *then-clause* statement is executed.

If more than one statement is used in a clause, grouping with { and } is required

  - Java syntax is like C/C++, but condition is Boolean type
  - Ada syntax allows use of multiple **elsif**'s to define nested conditions:

```
if<cond>then
  <statements>
elsif<cond>then
  <statements>
elsif<cond>then
  <statements>
...
else
  <statements>
end if
```

<u>Short-Circuited Conditions</u>

The purpose of the Boolean expression in a selection statement is not to compute a value to be stored, but to cause control to branch to various locations.

This observation allows us to generate particularly efficient code (called *jump code*) for expressions that are amenable to the short-circuit evaluation.

- Jump code is applicable not only to selection statements such as if. . . then . . . else, but to logically controlled loops as well.
- **Case/switch** statements are different from **if-then-else** statements in that an expression can be tested against multiple constants to select statement(s) in one of the *arms* of the **case** statement:
    - C, C++, and Java syntax:
      ```
      switch (<expr>)
      { case<const>: <statements>break;
        case<const>: <statements>break;
        ...
        default: <statements>
      }
      ```
    - **break** is necessary to transfer control at the end of an arm to the end of the **switch** statement

<u>Alternative Implementations</u>

A linear jump table is fast.
It is also space efficient when the overall set of case statement labels is dense and does not contain large ranges.
Sequential testing (as in an if. . . then . . . else statement) is the method of choice if the total number of case statement labels is small.
It runs in time *O*(*n*), where *n* is the number of labels.

<u>Syntax and Label Semantics</u>

As with if. . . then . . . else statements, the syntactic details of case statements vary from language to language.
Languages differ in whether they permit label ranges, whether they permit (or require) a default (else) clause, and in how they handle a value that fails to match any label at run time.
Iteration: It means the act of repeating a process usually with the aim of approaching a desired goal or target or result.
Each repetition of the process is also called an "iteration," and the results of one iteration are used as the starting point for the next iteration.

- A conditional that keeps executing as long as the condition is true.

e.g: while, for, loop, repeat-until, ...

- Iteration and recursion are the two mechanisms that allow a computer to perform similar operations repeatedly.

Enumeration-Controlled Loops
Enumeration controlled iteration originated with the do loop of Fortran I.

- **Fortran-IV:**

```
  DO 20 i = 1, 10, 2
  ...
20 CONTINUE
```

which is defined to be equivalent to

```
  i = 1
20 ...
  i = i + 2
  IF i.LE.10 GOTO 20
```

Pascal has simple design:

- o **for**<id> := <expr>**to**<expr> **do**<stmt>
  for<id> := <expr>**downto**<expr> **do**<stmt>
- o Can iterate over any discrete type, e.g. integers, chars, elements of a set
- o Index variable cannot be assigned and its terminal value is undefined

- C, C++, and Java do not have enumeration-controlled loops although the logically-controlled **for** statement can be used to create an enumeration-controlled loop:

Programmer's responsability to modify, or not to modify, i and n in loop body

C++ and Java also allow local scope for index variable.
for (**int** i = 1; i <= n; i++) ...
Code Generation for for Loops
Naively, the loop of can be translated as
r1 := first

```
r2 := step
r3 := last
L1: if r1 > r3 goto L2
. . .                         – – loop body; use r1 for i
r1 := r1 + r2
goto L1
L2:
```

<u>Semantic Complications</u>

- Use of an iteration count is fundamentally dependent on being able to predict the number of iterations before the loop begins to execute.
- In many languages, including Fortran and Ada, it is *not* possible in others, notably C and its descendants.
- If enumeration is only *one possible* purpose for the loop—the number of iterations or the sequence of index values may change as a result of executing the first few iterations.

<u>Generators in Icon:</u> Icon generalizes the concept of iterators, providing a *generator* mechanism. It causes any expression in which it is embedded to enumerate multiple values on demand.

<u>Problems with Enumeration-Controlled Loops:</u>

C/C++: This C program never terminates:

```
#include<limits.h>
main()
{ int i;
  for (i = 0; i <= INT_MAX; i++)
    ...
}
```

because the value of i overflows after the iteration with i==INT_MAX and i becomes a large negative integer

In C/C++ it is easy to make a mistake by placing a ; at the end of a **while** or **for** statement, e.g. the following loop never terminates:

```
i = 0;
while (i < 10);
{ i++; }
```

<u>Logically-Controlled Pretest Loops:</u>

- Logically-controlled *pretest* loops test an exit condition *before* each loop iteration.
- Not available Fortran-77 (!)
- Pascal:
    - **while**<cond>**do**<stmt>
      where the condition is a Boolean expression and the loop will terminate when the condition is false.

Multiple statements need to be enclosed in **begin** and **end**

Java is like C++, but condition is Boolean expression.
<u>Logically-Controlled Post test Loops:</u>

- Logically-controlled *post test* loops test an exit condition *after* each loop iteration
- Not available in Fortran-77 (!)
- C, C++:
  - o **do**<stmt>**while** (<expr>)
    where the loop will terminate when the expression evaluates to 0 and multiple statements need to be enclosed in { and }

Java is like C++, but condition is a Boolean expression

Logically-Controlled Mid test Loops:

- Logically-controlled *mid test* loops test exit conditions within the loop
- Ada:
  - o **loop**
      <statements>
    exit when<cond>;
      <statements>
    exit when <cond>;
      <statements>
    ...
    end loop
  - o Also allows exit of outer loops using labels:

Java is like C++, but combines Ada's loop label idea to allow jumps to outer loops.

<u>Combination Loops:</u>
Algol 60 provides a single loop construct that subsumes the properties of more modern enumeration and logically controlled loops.
It can specify an arbitrary number of “enumerators,” each of which can be a single value, a range of values similar to that of modern enumeration-controlled loops.
<u>Iterators:</u> A for loop iterates over the elements of an arithmetic sequence. We may wish to iterate over the elements of any well-defined set (*containers* or *collections*).

- Clu introduced an elegant *iterator* mechanism to do precisely that.

- Icon, provides a generalization of iterators, known as *generators*, that combines enumeration with backtracking search.

True Iterators: Clu, Python, Ruby, and C# allow any container abstraction to provide an *iterator* that enumerates its items.
The iterator resembles a subroutine that is permitted to contain yield statements, each of which produces a loop index value.
For loops are then designed to incorporate a call to an iterator.
Iterator Objects: As realized in most imperative languages, iteration involves both a special form of for loop and a mechanism to enumerate values for the loop.
These concepts can be separated.
Euclid, C++, and Java all provide enumeration-controlled loops reminiscent of those of Python.
An iterator is an ordinary object that provides methods for initialization, generation, testing etc.
Iterating with First-Class Functions
In functional languages, the ability to specify a function "in line" facilitates a programming idiom in which the body of a loop is written as a function, with the loop index as an argument. This function is then passed as the final argument to an iterator.
Iterating without Iterators: In a language with neither true iterators nor iterator objects, we can still decouple set enumeration from element use through programming conventions.
In C, for example, we might define a tree_iter type and associated functions that could be used in a loop as follows:

```
bin_tree *my_tree;
tree_iter ti;
...
for (ti_create(my_tree, &ti); !ti_done(ti); ti_next(&ti)) {
bin_tree *n = ti_val(ti);
...
}
ti_delete(&ti);
```

There are two principal differences between this code and the more structured alternatives: (1) the syntax of the loop is a good bit less elegant ,and (2) the code for the iterator is simply a type and some associated functions.

Recursion

When a function may directly or indirectly call itself.

- Can be used instead of loops
  – Functional languages frequently have no loops but only recursion

- Recursion can be less efficient, but most compilers for functional languages will optimize recursion and are often able to replace it with iterations

Iteration and Recursion

- Fortran 77 and certain other languages do not permit recursion.
- A few functional languages do not permit iteration.
- Most modern languages, however, provide both mechanisms.
- Iteration is in some sense the more "natural" of the two in imperative languages, because it is based on the repeated modification of variables.

- Recursion is the more natural of the two in functional languages, because it does *not* change variables.

Tail Recursive Functions

- *Tail recursive* functions are functions in which no computations follow a recursive call in the function.

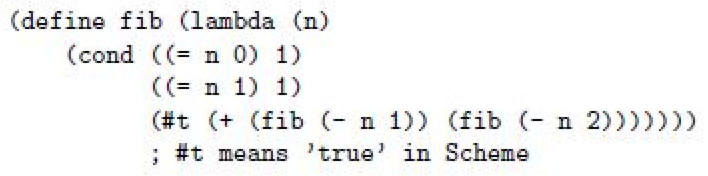
- A recursive call could in principle reuse the subroutine's frame on the run-time stack and avoid deallocation of old frame and allocation of new frame.

- This observation is the key idea to *tail-recursion* optimization in which a compiler replaces recursive calls by jumps to the beginning of the function.

For the gcd example, a good compiler will optimize the function into:

```
int gcd(int a, int b)
{ start:
   if (a==b) return a;
   else if (a>b) { a = a-b; goto start; }
   else { b = b-a; goto start; }
}
```

Thinking Recursively: Detractors of functional programming sometimes argue, incorrectly, that recursion leads to *algorithmically inferior* programs.
Fibonacci numbers, for example, are defined by a mathematical recurrence
The naive way to implement this recurrence in Scheme is



```
(define fib (lambda (n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (#t (+ (fib (- n 1)) (fib (- n 2))))))
          ; #t means 'true' in Scheme
```

Applicative- and Normal-Order Evaluation
It is possible to pass a representation of the *unevaluated* arguments to the subroutine instead, and to evaluate them only when (if) the value is actually needed.
The former option (evaluating before the call) is known as *applicative-order evaluation*;the latter (evaluating only when the value is actually needed) is known as *normal order evaluation*.
It also occurs in short-circuit Boolean evaluation, *call-by-name* parameters, and certain functional languages .
Lazy Evaluation
Scheme provides for optional normal-order evaluation in the form of built-in functions called delay and force.
These functions provide an implementation of *lazy evaluation*.
A delayed expression is sometimes called a *promise*.
The mechanism used to keep track of which promises have already been evaluated is sometimes called *memoization*.
Continuation-Passing-Style: Even functions that are not tail-recursive can be optimized by compilers for functional languages by using *continuation-passing style:*

With each recursive call an argument is included in the call that is a reference to the remaining work.

Other Recursive Function Optimizations: Another function optimization that can be applied by hand is to remove the work after the recursive call and include it in some other form as an argument to the recursive call

- For example:

```
typedefint (*int_func)(int);
int summation(int_func f, int low, int high)
{ if (low==high) return f(low)
  elsereturn f(low)+summation(f, low+1, high);
}
```

  can be rewritten into the tail-recursive form:

```
int summation(int_func f, intlow, int high, int subtotal)
{ if (low==high) return subtotal+f(low)
  elsereturn summation(f, low+1, high, subtotal+f(low));
}
```

- **Nondeterminacy:**

Our final category of control flow is nondeterminacy.
A nondeterministic construct is one in which the choice between alternatives is deliberately unspecified.
In most languages, operator or subroutine arguments may be evaluated in any order.
Some languages, notably Algol 68 and various concurrent languages, provide more extensive nondeterministic mechanisms, which cover statements as well.