MODULE 1

Programming Language Pragmatics 3rd Edition

Michael L. Scott

Soumya Mathew, AP,VJCET

Introduction

- Why programming languages?
  - Describe algorithms & data structures
  - To specify, organize and reason about the various aspects of problem solving.

Designers of pgming languages have 2 main goals:

- Making computing convenient for people
- Making efficient use of computing machines

Name, Scope, and Binding

- A name is exactly what you think it is
  - Most names are identifiers
  - symbols like + or :=
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program in which the binding is active

NAMES

- A name is a mnemonic character string used to represent something meaningful.
- allow us to refer to variables, constants, operations, types, and so on using symbolic identifiers.
- Provides abstraction, so that a programmer can associate a name for a complicated program segment.

SCOPES

- It is the range of statements in which the variable is visible.
- The scope rules of a language determine how a name is associated with a variable or how a name is associated with an expression.
- A variable has a local scope, if it is declared within a program unit or block.
- A variable is nonlocal in a program, if it is declared outside the program unit or block.

Binding

- Binding Time is the time at which a binding is created or, more generally, the point at which any implementation decision is made.
  - language design time
    - program structure, possible data types
  - language implementation time
    - I/O, arithmetic overflow, type compatibility
  - program writing time
    - algorithms, names and data structures

Binding

- compile time
  - Compilers map high level constructs to machine code
- link time
  - layout of whole program in memory by linking different modules of a program.
- load time
  - choice of physical addresses- loads the program into memory.
- run time

- *It* covers the entire span from the beginning to the end of execution. Bindings of values to variables occur at run time.

Binding

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively.

Binding

- In general, early binding times are associated with greater efficiency
- Later binding times are associated with greater flexibility
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times

SCOPE RULES

- Scope Rules - control bindings
  - Fundamental to all programming languages is the ability to name data, i.e., to refer to data using symbolic identifiers rather than addresses
  - Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names.

Scope Rules

- A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted.
- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)
  - make references to variables

Scope Rules

- Static Scoping
- Nested Subroutines
  - Access to nonlocal objects
- Declaration Order
  - Declarations and Definitions
  - Nested Blocks
- Modules
  - Encapsulating data and subroutines
  - Modules as abstractions
  - Imports and Exports
  - Modules as Managers
- Module Types and Classes
  - Object Orientation
  - Modules containing classes
- Dynamic Scoping

1.Static scope

- With STATIC SCOPE RULES, a scope is defined in terms of the physical structure of the program.
  - The determination of scopes can be made by the compiler
  - All bindings for identifiers can be resolved by examining the program
  - Typically, we choose the most recent, active binding made at compile time.
  - Most compiled languages, C and Pascal included, employ static scope rules

Static scope

- Initially, simple versions (such as early Basic) had only 1 scope - the entire program.
- Later, distinguished between local and global (such as in Fortran).
- In general, a local variable only "exists" during the single execution of that subroutine. So the variable is recreated and reinitialized each time it enters that procedure or subroutine.
- Eg) variables in C created in a loop, and which don't exist outside the loop.

Static scope

- Consider the example in C

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

2.Nested Subroutines

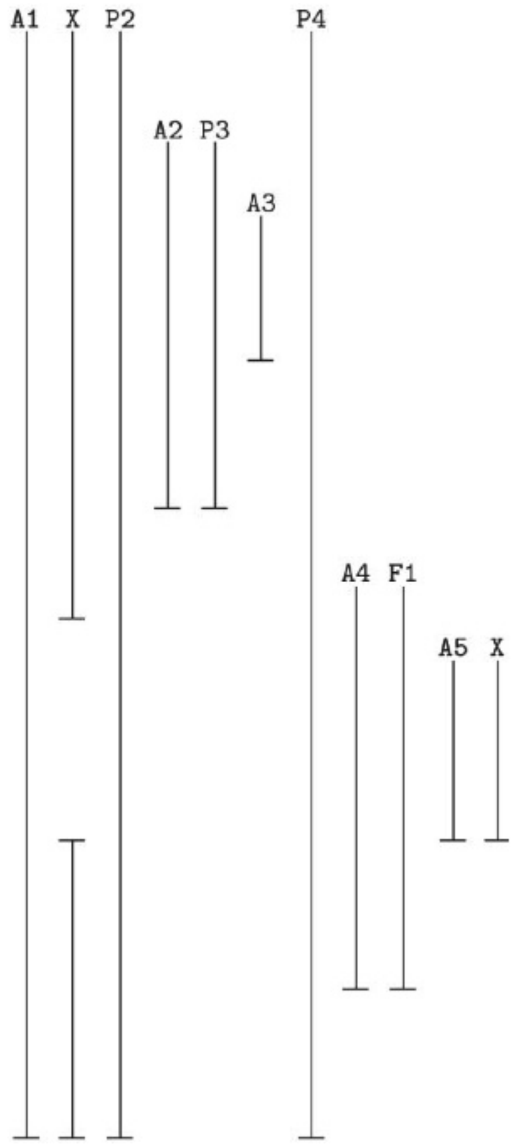
- Nested subroutine concept was first introduced in Algol 60.
- Later, it is a feature of many modern languages, including Pascal, Ada, ML, Python, Lisp and FORTRAN 90.
- C and its descendants allow nested classes and scopes.

Closest nested scope rule

- The classical example of static scope rules is the most closely nested rule used in block structured languages such as Algol 60 and Pascal
  - An identifier is known in the scope in which it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope
  - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found

Nesting in Pascal

```
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
```



Access to non-local objects

- If a subroutine is declared at the outermost nesting level of the program, then its access is throughout the program. so there is no need of static link at run time.
- If a subroutine is nested k levels deep, then its static link and those of its parent, grandparent and so on and it forms a static chain of length k at run time.

Scope Rules

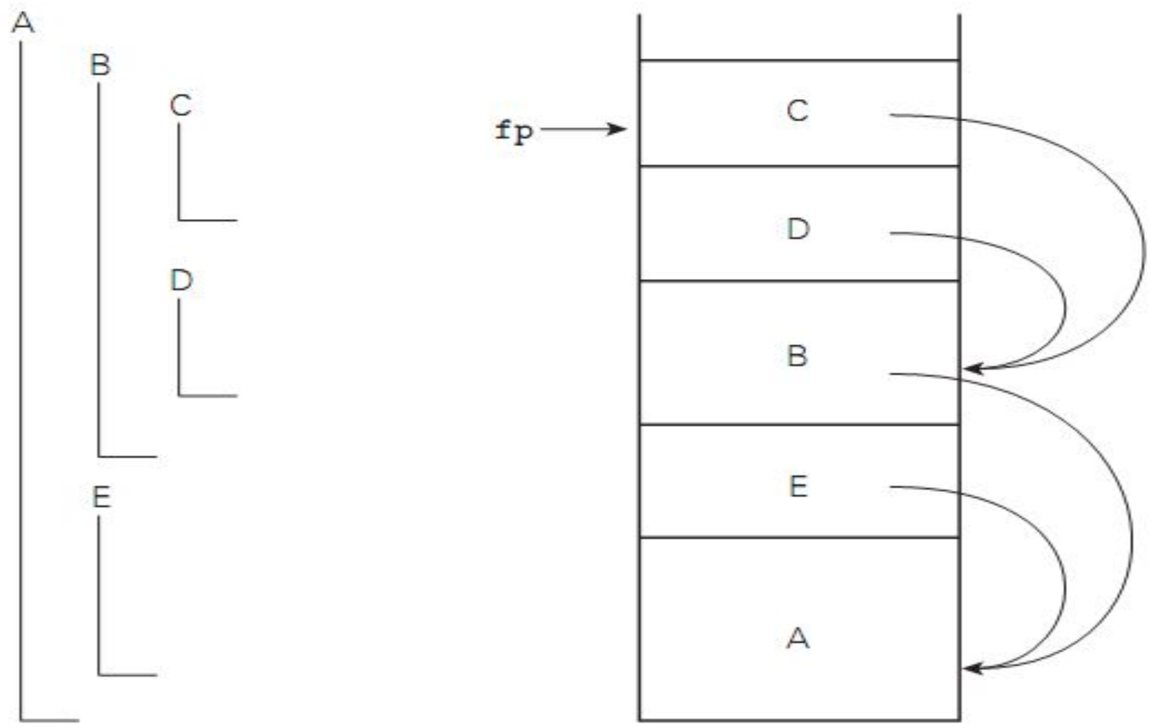


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

3.Declaration order

- Declaration order can be a key concept: Does an object exist in its scope before it is created?
- depends on the language.
- Example (in Pascal):

```
const N = 10;
…
procedure foo;
const
   M = N; (* semantic error because of next line*)
   …
   N = 20;
```

  - Compiler will complain that N is being used “before its declaration” on line 5.

Declaration order

- In pascal, the scope of a variable declaration is only within a function. Also, it should be declared before using it in any expressions.
- If the same program segment had been written in C,C++ or Java, no semantic errors would be reported.
- The declaration of M would refer to the first (outer) declaration of N. so it is called as a **whole-block scope** in C#.

Declaration order

- An example in C++:

```
Class Contact
{
   public int age;
   public void F()
   {
   age=18;
   }
public void G()
   {
   int age;
   age=21;}}
```

Declaration order

- Here the field ‘age’ is in scope throughout the entire body including within F and G and F uses tat.
- In G, the scopes overlap so that the name declared in the outer scope is hidden by the inner scope.
- Python allows both global and local declarations.
- In Scheme, which is a functional programming language that shares similar characteristics of Lisp.
- it allows scope as whole block and declaration-to-end-of -block scopes.

Eg)
```
(let ((A 1))        ;outer scope, with A defined to be 1
    (let ((A 2)     ;inner scope, with A defined to be 2
        (B A))      ; and B defined to be A
        B))         ;return the value of B
```

- Here the nested declarations of A and B don't take effect until after the end of the declaration list.ie the final value of B is defined to be the outer A and so the code returns the value as 1.

Declarations and Definitions

- Declaration introduces a name and indicates its scope, but may omit certain implementation details.
- A definition describes the object in sufficient detail for the compiler to determine its implementation.
- If a declaration is not complete enough to be a definition, then a separate definition must appear somewhere else in the scope.

Declarations and Definitions

```
struct manager;             /* declaration only */
struct employee {
struct manager *boss;
struct employee *next_employee;
 ...
};
struct manager {               /* definition */
struct employee *first_employee;
...
};
```

Nested Blocks

- In many languages, including Algol 60, C89, and Ada, local variables can be declared not only at the beginning of any subroutine, but also at the top of any begin...end({...})block.
- Other languages, including Algol68,C99,and all of C’s descendants, allowing declarations wherever a statement may appear.
- In most languages a nested declaration hides any outer declaration with the same name.
- If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the point of declaration by the inner block.

4.Modules

- A module is a part of a program.
- Programs are composed of one or more independently developed modules that are not combined until the program is linked.
- Programs are divided into modules
  - to reduce the effort of programmers
  - Information hiding

Encapsulating data and subroutines

- Static variables allow a subroutine to have "memory"—to retain information from one invocation to the next.
- Thus, static variables allow programmers to build single-subroutine abstractions.
- Eg) stack abstraction - hide the representation of the stack—its internal structure—from the rest of the program, so that it can be accessed only through its push and pop routines.

Modules as abstractions

- A module allows a collection of objects such as subroutines, variables, types, and so on and it will be encapsulated in such a way that

(1) objects inside are visible to each other.

(2) objects on the inside are not visible on the outside unless explicitly exported.

(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 in restricted ways.
- Variables or types may be exported, meaning 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 into which names must be explicitly imported are said to be closed scopes.
- Modules that do not require imports are said to be open scopes.

Imports and Exports

Eg) In Ada,Java,C# etc ,a name ‘add’ exported from module A is automatically visible in peer module B as A.add. It becomes visible as merely add if B explicitly imports it.

Modules as Managers

- Each module defines a single abstraction.
- 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.
- It is possible to import variables and export functions/procedures between different modules.
- Consider the program structure in modula-2, where a module acts as manager by importing and exporting variables and procedures.

Module as Managers

```
CONST stack_size = ...
TYPE element = ... ...
MODULE stack_manager;
IMPORT element, stack_size;
EXPORT init_stack, push, pop;
    PROCEDURE init_stack(......)
    BEGIN
        ..............
    END init_stack;
PROCEDURE push(............)
BEGIN
    ................
END push;
    PROCEDURE pop(......)
    BEGIN
        ..........
    END pop;
END stack_manager;
```

OUTPUT

```
var A, B : stack;
var x, y : element;
 ...
init_stack(A);
init_stack(B);
...
 push(A, x);
...
y := pop(B);
```

5.Module Types and Classes

- In a Module Type, the programmer can declare arbitrary number of similar objects.
- The difference between module as manager and module as type is that,
- In module as type, the programmer consider a module's subroutine itself belonging to the stack type as (A.push(x)).
- In module as manager, the programmer consider a module's subroutine as outside entities to which the stack can be passed as an argument.

  eg) (push(A,x)) .

Module types and Classes

- In module as manager, there is a separate pair of push and pop operations for every stack. so it would be wasteful to create multiple copies of the code.
- In module as types, any arbitrary number of stacks share a single pair of push and pop operations and the compiler arranges a pointer to the relevant stack to be passed as a hidden parameter.

```
const stack_size := ...
 type element : ... ...
 type stack = module
        imports (element, stack_size)
       exports (push, pop)
type
       stack_index = 1..stack_size
var
       s : array stack_index of element
       top : stack_index
procedure push(elem : element) = ...
function  pop  returns  element = ...
 ...
initially
       top := 1
 end stack
```

OUTPUT

```
var A, B : stack
var x, y : element
...
A.push(x)
 ...
y := B.pop
```

Object orientation

- As an extension to module-as-type, many languages now provide a class construct for object-oriented programming.
- classes can be thought of as module types that have been augmented with an inheritance mechanism.
- Inheritance allows new classes to be defined as extensions or refinements of existing classes.
- Inheritance allows all or most operations belonging to a class, and in which new class can inherit most of their operations from existing class, without the need to rewrite code.

Modules containing classes

```
class  DerivedClassName(BaseClassName):
    <statement-1>
        . . .
    <statement-N>
```

If the base class is defined in a separate module,

```
class DerivedClassName(modname.BaseClassName):
```

Dynamic Scoping

- In dynamic scoping, the bindings between names and objects depend on the flow of control at runtime, ie, the order in which subroutines are called.

Eg. Languages like APL, Snobol,early versions of Lisp.

- All type of syntax and semantic checking, type checking in expressions, argument checking in subroutine calls are deferred until run time.
- To do all these checks, languages with dynamic scoping tend to be interpreted, rather than compiled.

Dynamic Vs Static

- The key idea in **static scope rules** is that bindings are defined by the physical structure of the program before run time.
- With **dynamic scope rules**, bindings depend on the current state of program execution.

Lifetime and Storage Management

- Key events
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects

Lifetime and Storage Management

- The period of time from creation to destruction is called the LIFETIME of a binding.
- Similarly, the time between the creation and destruction of an object is the object’s lifetime
  - If object outlives binding it's garbage
  - If binding outlives object it's a dangling reference .(a binding to object that is no longer live).
- The textual region of the program in which the binding is *active* is its scope

Lifetime and Storage Management

- Storage Allocation mechanisms
  - Static allocation
  - Stack based allocation
  - Heap based allocation
  - Garbage collection

Static Allocation

- Static objects are given an absolute address that is retained throughout the program's execution.

Static allocation for
- global variables
- local variables
- Numeric & string-valued constant literals.
- explicit constants.
- scalars may be stored in the instructions.

It's a fixed allocation of memory, cannot increase or decrease the size of memory blocks.

Static Allocation

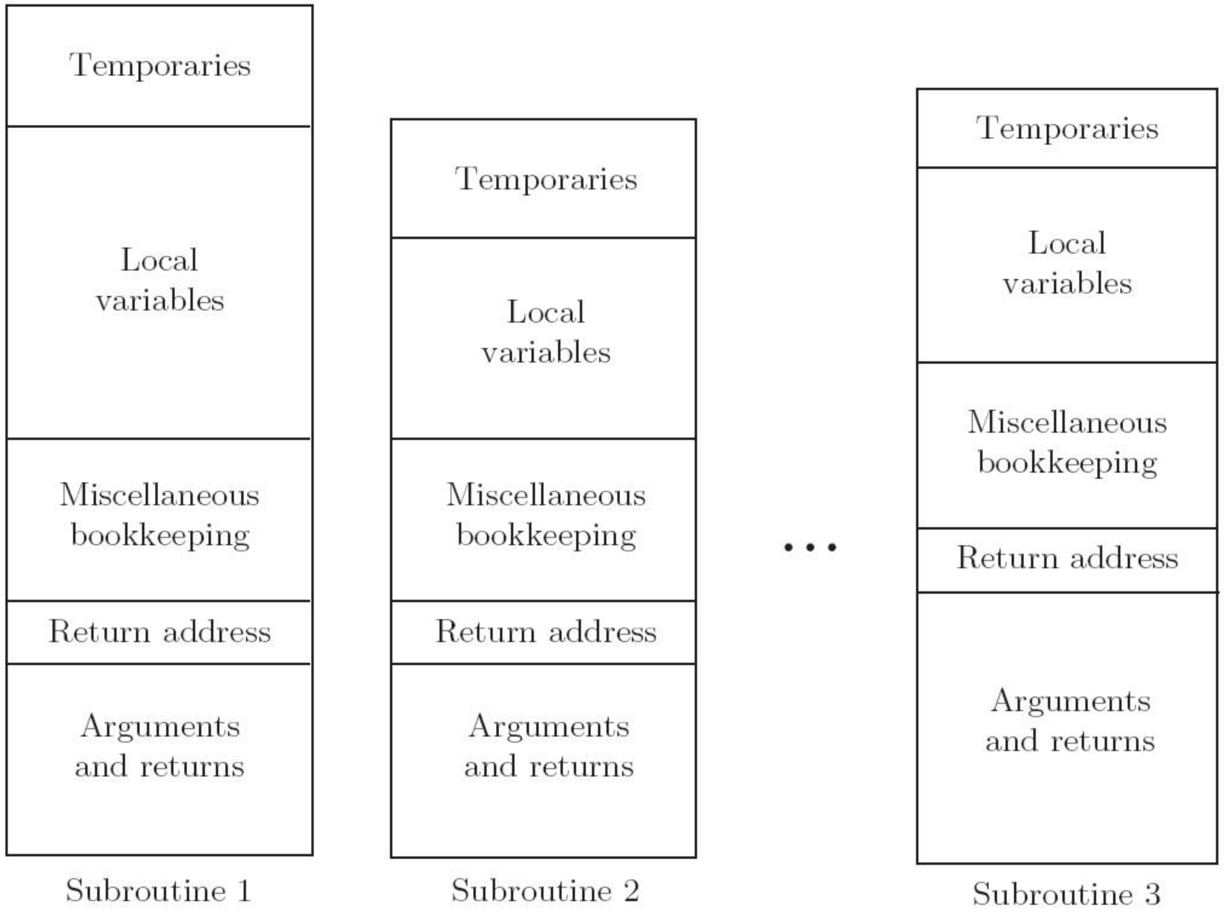


Figure 3.1: **Static allocation of space for subroutines in a language or program without recursion.**

Stack based allocation

- Stack objects are allocated and deallocated in last-in,first-out order, usually in conjunction with subroutine calls and returns.
- Central stack for
  - parameters
  - local variables
  - temporaries
- Why a stack?
  - allocate space for recursive routines
  - reuse space

Stack based allocation

- Contents of a stack frame.
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, reference to stack pointer of called subrotine, debugging info etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Stack based allocation

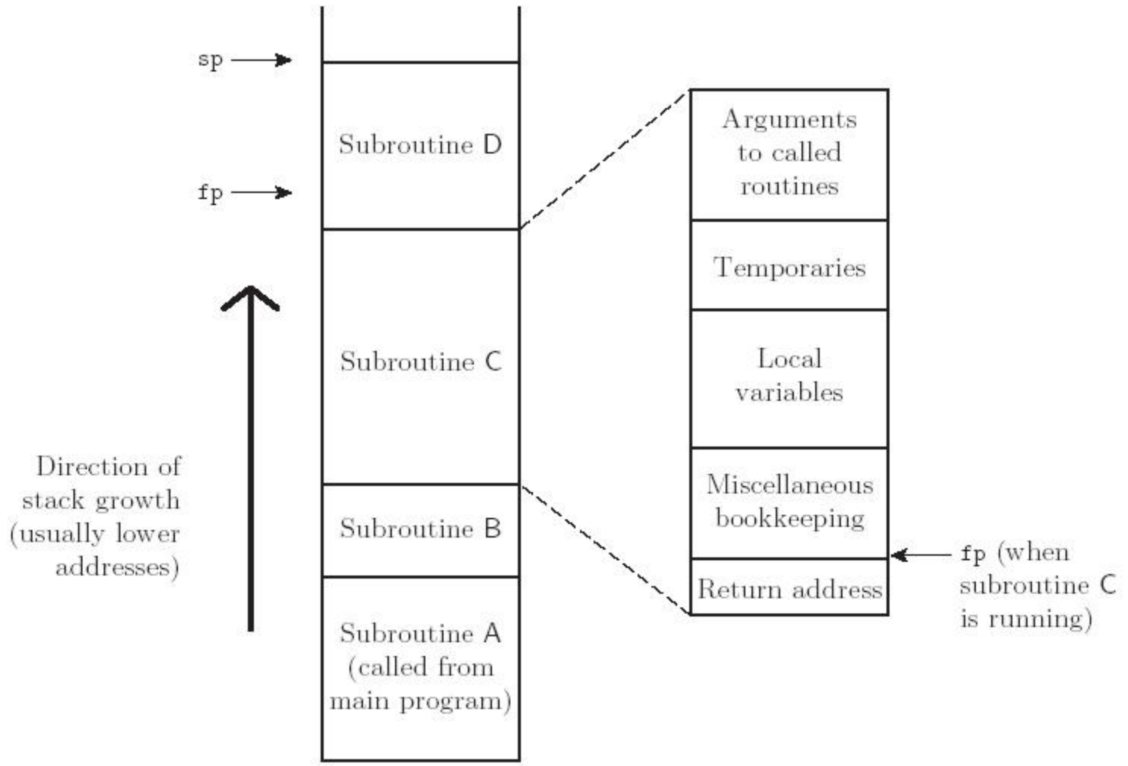

sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine A (called from main program)
Direction of stack growth (usually lower addresses)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)

Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Stack based allocation

- Maintenance of the stack is the responsibility of the subroutine’s
  - calling sequence—the code executed by the caller immediately before and after the call
  - prologue (code executed at the beginning)
  - epilogue (code executed at the end)

Heap based allocation

- Heap objects may be allocated and deallocated at arbitrary times.
- 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
- objects like fully general character strings,lists,and sets, whose size may change as a result of an assignment statement or other update operation.

Heap based allocation

- Heap for dynamic allocation

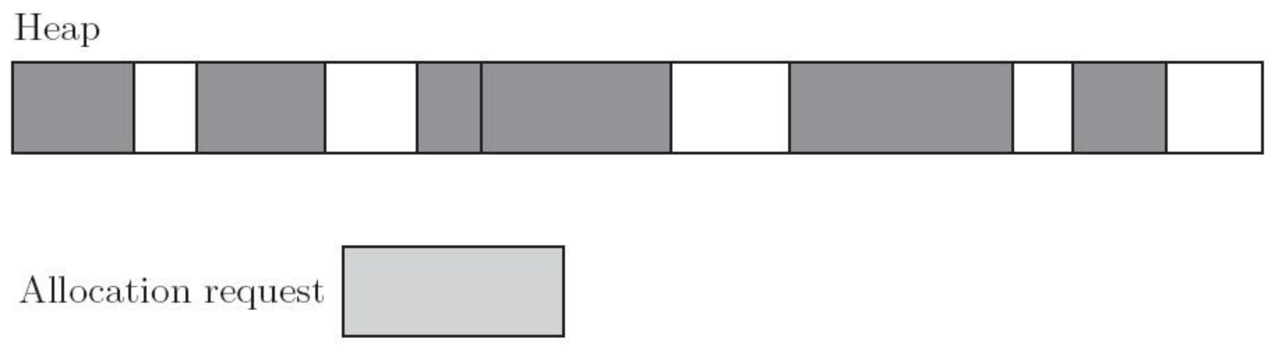


Figure 3.3: **External fragmentation.** The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Garbage Collection

- Many languages specify that objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable.
- The run-time library for such a language must then provide a **garbage collection** mechanism to identify and reclaim unreachable objects.
- Deallocation errors are most common.
- Designers and developers choose automatic garbage collection as an essential feature.
- Garbage collection algorithms are improved.
- it is the automatic recycling of dynamically allocated memory.

Binding of Referencing Environments

- REFERENCING ENVIRONMENT of a statement at run time is the set of active bindings
- A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding.
- The reference environment of a statement is the collection of all names that are visible in the statement.
- Referencing environments are generally invariant during program execution, i.e., values may change but name associations do not

Binding of Referencing Environments

- 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 runtime.

Binding of Referencing Environments

Deep and Shallow Binding

```
def sub1():
    x = 1
    def sub2():
        print x
    def sub3():
        x = 3
        sub4(sub2)
    def sub4(f):
        x = 4
        f()
    sub3()
```

Deep and Shallow Binding

- Deep binding : **Takes the environment of the parent function.** So in the example, no matter which sub-function gets called the value of x would be **1**.
- Shallow binding : **Takes the environment of the "final" calling function.** Here, ultimately the **final** function to call **sub2** is **sub4**, so **sub2** would take the value of **x** initialized in **sub4** and would print **4**.

Deep and Shallow Binding

- **Deep binding** binds the environment at the time a procedure is passed as an argument. so it is called as **early binding** of the referencing environment
- **Shallow binding** binds the environment at the time a procedure is actually called. so it is called as **late binding** of the referencing environment.
- Deep Binding is to use the environment of the definition of the passed subprogram. Most natural for static-scoped languages.
- Shallow Binding is to use the environment of the call statement that CALLS the passed subprogram. Most natural for dynamic-scoped languages.

Deep and Shallow Binding

```
function f1()
{
var x = 10;
        function f2(fx)
        {
                var x;
                x = 6;
                fx();
        };
        function f3()
        {
                print x;
        };
        f2(f3);
};
```

Deep and Shallow Binding

- **Deep binding.**

Here *f3()* gets the environment of *f1()* and prints the value of *x* as *10* which is local variable of *f1()*.

- **Shallow binding.**

f3() is called in *f2()* and hence gets the environment of *f2()* and prints the value of x as 6 which is local to *f2()*

1.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.
- A subroutine closure contains
  - A pointer to the subroutine code
  - The current set of name-to-object bindings

Subroutine closures

- In a language with dynamic scoping, referencing environment depends on

(1) the use of a stack (*association list*) for all active variables

(2) keep a central reference table for run time lookup of names.

Subroutine closures

- A closure in a language with static scoping captures the current instance of every object, at the time the closure is created.
- When the subroutine is called, it will find these captured instances, even if newer instances have subsequently been created by recursive calls.

First-class values and unlimited extent

- In general, 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.
- Eg) integers and characters are first class values in most programming languages.
- By contrast, a “second-class” value can be passed as a parameter, but not returned from a subroutine or assigned into a variable Eg) arrays in C/C++
- a “third-class” value cannot even be passed as a parameter eg)labels of goto-statements.

First-class values and unlimited extent

- labels are third-class values in most programming languages, but second-class values in Algol.
- subroutines are first-class values in all functional programming languages and most scripting languages.
- They are also first-class values in C# and, with some restrictions, including Fortran, Modula-2 and -3,Ada 95, C, and C++.11 They are second-class values in most other imperative languages, and third-class values inAda 83.

First-class values and unlimited extent

- If local objects were destroyed and their space reclaimed at the end of each scope’s execution, then the referencing environment captured in a long-lived closure might become full of dangling references.
- To avoid this problem, most functional languages specify that local objects have **unlimited extent**: their lifetimes continue indefinitely.
- Their space can be reclaimed only when the garbage collection system is able to prove that they will never be used again.

First-class values and unlimited extent

- Local objects in most imperative languages have limited extent: they are destroyed at the end of their scope’s execution.
- Space for local objects with limited extent can be allocated on a stack. Space for local objects with unlimited extent must generally be allocated on a heap.

3.Object Closures

- In object-oriented languages, we can encapsulate the subroutine as a method of a simple object, and let the object’s fields hold context for the method.
- 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.

Control Flow

- Basic paradigms for control flow:
  - Sequencing
  - Selection
  - Iteration
  - Procedural Abstraction
  - Recursion
  - Concurrency
  - Exception Handling and Speculation
  - Nondeterminacy

Control Flow

- Basic paradigms for control flow:
  - Sequencing: order of execution
  - Selection (also alternation): generally in the form of if or case statements
  - Iteration: loops
  - Recursion: expression is defined in terms of (simpler versions of) itself
  - Nondeterminacy: ordering or choice among statements is deliberately left unspecified.

Expression Evaluation

- An expression generally consists of either a simple object(e.g.,a literal constant,or a named variable or constant)or an operator or function applied to a collection of operands or arguments,each of which in turn is an expression.
- A function call

  my_func(A, B, C)
- A typical operators

  a+b –c

In Ada, for example, a+b is short for "+"(a, b);

Expression Evaluation

- In general, a language may specify that function calls employ prefix, infix, or postfix notation.
  - prefix: op a b or op(a, b) or (op a b)
  - infix: a op b
  - postfix: a b op
- Conditional Expressions

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 as

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

Expression Evaluation

- Most imperative languages use infix notation for binary operators and prefix notation for unary operators and functions.
- Lisp uses a variant of prefix notation for all functions, which is known as **Cambridge Polish notation**, it places the function name inside the parentheses:

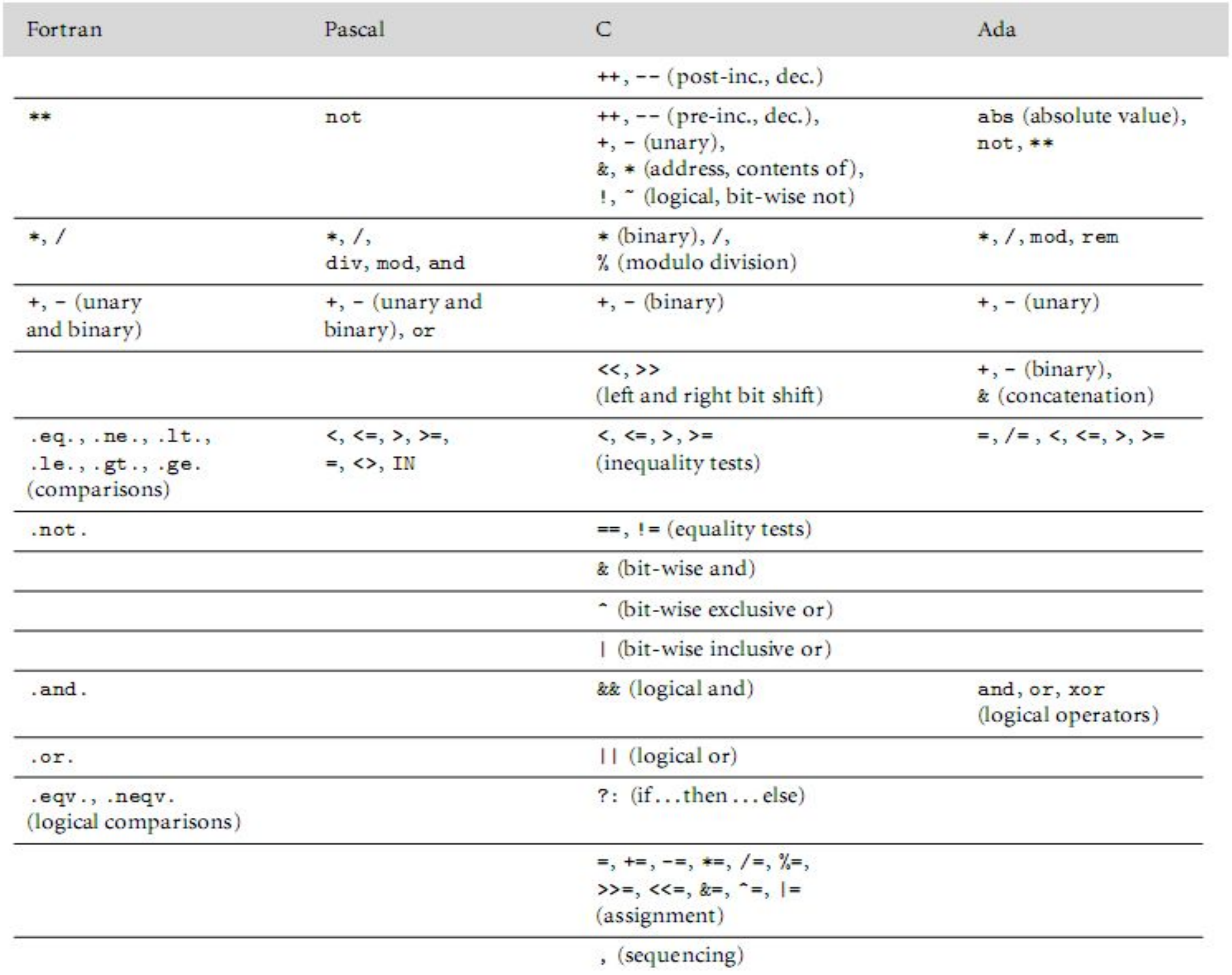
Eg) (* (+ 1 3) 2) ; that would be (1 + 3) * 2 in infix.

(- (+ 1 (* 2 3)) 4); that would be 1 + 2 * 3 – 4

1.Precedence & Associativity

- In any language, the choice among alternative evaluation orders depends on the precedence and associativity of operators.
- C has 15 levels - too many to remember
- Pascal has 3 levels - too few for good semantics
- Fortran has 8
- Ada has 6

Precedence & Associativity



Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /= , <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Precedence & Associativity

- In most languages arithmetic operators have higher precedence than relational operators, which in turn have higher precedence than the logical operators.
- Exceptions include APL and Smalltalk, in which all operators are of equal precedence; parentheses must be used to specify grouping.
- Associativity rules specify that the basic arithmetic operators almost always associate left-to-right.

Eg) 9-3-2 is 4 and not 8.

2.Assignments

- Each assignment takes a pair of arguments: a value and a reference to a variable into which the value should be placed.
- Imperative programming languages are described as **“computing by means of side effects”. Eg)C#,python ,Ruby**
- ie,in such languages an evaluation of an assignment within an expression or statement results in a value that may change the result of any later computations in which the variable appears.
- But **functional languages have no side effects**,because the value of expression depends only on the referencing environment in which the expression is evaluated, not on the time at which the evaluation occurs. So expressions in a purely functional languages are said to be **referentially transparent**.Eg) Haskell and Miranda are purely functional.

2.1 References and values

- C language uses ***Value model*** of variables.

- L- values : its on the left-hand side of assignment statements.(expressions that denote locations)
- R-values : its on the right-hand side of assignment statements.(expressions that denote values)
- Not all expressions can be l-values, because not all values have a location, and not all names are variables.

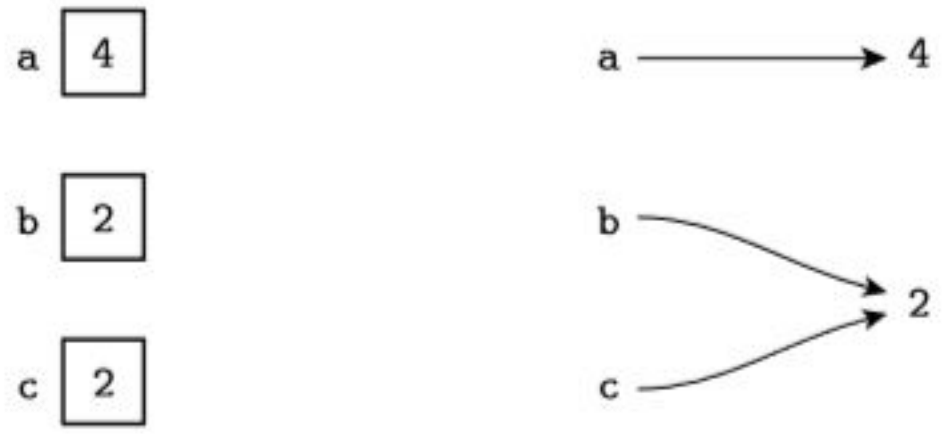
2.1 References and values

- Languages include Algol 68, Lisp/Scheme, ML, Haskell, and Smalltalk uses a ***reference model*** of variables.
- ie, a variable is not a named container for a value; rather, it is a named reference to a value.
- Eg) In Pascal

```
b := 2;
c := b;
 a := b + c;
```

- We put the value 2 in b and then copy it into c. We then read these values, add them together, and place the resulting 4 in a.

2.1 References and values



Here it represents value model and reference model of variables.

With a value model, any integer variable can contain the value 2

With a reference model, ony one 2 to which any variable can refer.

2.2 Boxing

- In certain languages like,recent versions of java perform automatic boxing and unboxing operations that avoid the need of wrapper class.
- Converting a primitive value into an object of the corresponding wrapper class is called **autoboxing**.

  Eg) converting int to Integer class.
- Converting an object of a wrapper type to its corresponding primitive value is called **unboxing**.

  Eg) conversion of Integer to int.

2.3 Orthogonality

- Features that can be used in any combination
- Algol 68 is primary example. Here, everything is an expression (and there is no separate notion of statements).
  - Example:

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

2.4 Combination Assignment Operators

- To eliminate the compile-time or run-time cost of redundant address calculations, and to avoid the issue of repeated side effects, many languages, beginning with Algol68, and including C and its descendants, provide assignment operators to update a variable.

Eg) statements like a=a+1 can be written as a+=1;

b.c[3].d = b.c[3].d * e; can be written as b.c[3].d * =e;

A[--i]=b;

2.5 Multiway Assignment

- Some languages (including ML, Perl, Python and Ruby) allow multiway assignment.
  - Example: a,b = c,d;
  - Defines a tuple; equivalent to a = c; b=d;

  Eg) a, b = b, a; //swap a and b

3.Initialization

- 3 main reasons why initialization is useful.

  1. a static variable that is local to a subroutine needs an initial value in order to be useful.

  2. For any statically allocated variable, an initial value that is specified in the declaration can be preallocated in global memory by the compiler, avoiding the cost of assigning an initial value at run time.

  3. Accidental use of an uninitialized variable is one of the most common programming errors.

3.Initialization

Dynamic Checks

- Its a semantic error checking during run time to catch the errors such as uninitialized variables, program bug that is masked by a default value etc.

Definite Assignment

- For local variables, java and C# define a notion of definite assignment that makes impossible the use of uninitialized variables.
- It is based on the control flow of the program and can be statically checked by the compiler.
- Every possible control path to an expression must assign a value to every variable in that expression.

4.Ordering within Expressions

- There are two main reasons why the order can be important:

1. Side effects:

- – often discussed in the context of functions
- – a side effect is some permanent state change caused by execution of function.
  - some noticable effect of call other than return value

Eg) a-f(b)-c*d

Here 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.

4.Ordering within Expressions

2. Code improvement

- There is no specific order for the evaluation of operands and arguments that are undefined. so the compiler can choose whatever order results in faster code.
- Eg) a:=B[i];

```
        c:=a*2+d*3;
```

Here it is desirable to evaluate d *3 before evaluating a*2 because the previous statement, a:=B[i] will need to load a value from memory. Thus code improvement leads to proper processor utilization.

4.Ordering within Expressions

Applying Mathematical Identities

Some language implementations (e.g Fortran) allow the compiler to rearrange expressions involving operators whose mathematical abstractions are commutative,associative,and/or distributive,in order to generate faster code.

Applying Mathematical Identities

Consider the example in Fortran,

```
a=b+c
d=c+e+b
```

Some compilers will rearrange this as

```
a=b+c
d=b+c+e
```

They can then recognize the common subexpression in the first and second statements, and generate code equivalent to

```
a=b+c
d=a+e
```

5.Short-Circuit Evaluation

- Boolean expressions provide a special and important opportunity for code improvement and increased readability.
  - Consider `(a < b) && (b < c)`:
  - If `a >= b` there is no point evaluating whether `b < c` because `(a < b) && (b < c)` is automatically false.
- A compiler that performs short-circuit evaluation of Boolean expressions will generate code that skips the second half of both of these computations when the overall value can be determined from the first half.

Example in C

```
p = my_list;
while (p && p->key != val)
p = p->next
```

p->key will be accessed only if p is not null

Example in Pascal

- Pascal donot support short circuit evaluation.
- Pascal does not short-circuit and
- and or

```
p := my_list;
while (p <> nil) and (p^.key <> val) do
p := p^.next;
```

- both of the <> relations will be evaluated before and-ing their results together

Solution in Pascal

- Pascal programmer introduce an **auxiliary Boolean variable**

```
p := my_list;
still_searching := true;
while still_searching do
if p = nil then
still_searching := false
else if p^.key = val then
still_searching := false
else
p := p^.next;
```

- Short-circuit evaluation is used to avoid **out-of-bound** subscripts

```
const MAX = 10;
int A[MAX]; /* indices from 0 to 9 */
...
if (i >= 0 && i < MAX && A[i] > foo) ...
```

division by zero:

```
if (d <> 0 && n/d > threshold) ...
```

When not to use Short Circuit Evaluation

- Short circuiting is not good for situations in which a Boolean subexpression can cause a side effect

When not to use Short Circuit Evaluation

```
function tally(word : string) : integer;
(* Look up word in hash table. If found, increment tally; If not
found, enter with a tally of 1. In either case, return tally. *)
...
function misspelled(word : string) : Boolean;
(* Check to see if word is mis-spelled and return appropriate
 indication. If yes, increment global count of mis-spellings. *)
...
 while not eof(doc_file) do begin
w := get_word(doc_file);
 if (tally(w) = 10) and misspelled(w) then
writeln(w)
end;
writeln(total_misspellings);
```

- Ada uses Regular Boolean Operators: and,or
- Short Circuit Boolean operators: and then, or   else
- C
- Non- Short Circuit Boolean operators: & and |
- Short Circuit Boolean operators: &&, ||

STRUCTURED AND UNSTRUCTURED FLOW

- Control flow in assembly languages is achieved by means of conditional and unconditional jumps.
- Early versions of Fortran used goto statements for most nonprocedural control flow:

```
    if (A .lt. B) goto 10

    .........

    10
```

- The 10 on the bottom line is a statement label.
- Goto statements also featured prominently in other early imperative languages.

1.STRUCTURED ALTERNATIVES TO goto

- Once a goto might have been used to jump to the **end of the current subroutine**, most modern languages provide an explicit **return** statement.
- 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.

1.1 Multilevel Returns

- Returns and gotos allow control to return from the current subroutine.

1.STRUCTURED ALTERNATIVES TO goto

1.2 Errors and Other Exceptions

- Conditions that require a program to “**back out**” are usually called **exceptions**.
- many modern languages provide an exception handling mechanism for convenient, nonlocal recovery.
- programmer appends a block of code called a **handler** to recover from the exception.
- Common Lisp and Ruby provide mechanisms for both multilevel returns and exceptions.

2. CONTINUATIONS

- **continuation** is an abstract representation of the control state of a program.
- continuation consists of a code address and a referencing environment to be restored when jumping to that address.
- They also appear as first-class values in certain languages (Scheme and Ruby).
- A continuation is similar to a GOTO in that it transfers control from an arbitrary point in the program to a previously marked point.
- A continuation is more flexible than GOTO in those languages that support it, because it can transfer control out of the current function, something that a GOTO cannot do in most structured programming languages.
- Eg) continue statements in C language.

SEQUENCING

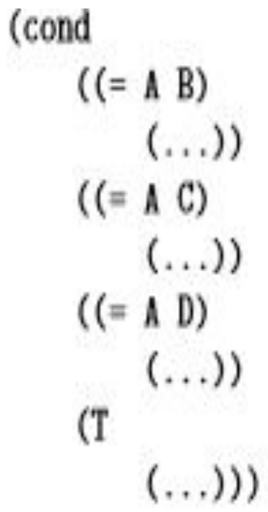
  - specifies a linear ordering on statements
    - one statement follows another
    - the first statement executes before the second.
    - Blocks of code are executed in a sequence.
    - Block are generally indicated by { … } or similar construct.
- In languages like Algol 68, the value of a an expression is the value of its final element.
- In Common Lisp, the programmer can choose to return the value of the first element, the second, or the last.
- They simplify code improvement, by permitting the safe rearrangement of expressions.

SELECTION

- Selection: introduced in Algol 60
  - sequential if statements

```
if ... then ... else
  if ... then ... elsif ... else
```

  - Lisp variant:



```
(cond
    ((= A B)
        (...))
    ((= A C)
        (...))
    ((= A D)
        (...))
    (T
        (...)))
```

SELECTION

1.Short-circuited Conditions

- 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 will generate a particularly efficient code called **jump code** for expressions.

SELECTION

- jump code
  - for selection and logically-controlled loops.
  - no point in computing a Boolean value into a register, then testing it
  - instead of passing register containing Boolean value as an attribute, pass inherited attributes INTO expression indicating where to jump to if true, and where to jump to if false.

SELECTION

- Jump is especially useful in the presence of short-circuiting

```
 if ((A > B) and (C > D)) or (E <> F)
   then
        then_clause
    else
        else_clause
```

Pascal(does not use short-circuit evaluation)

```
     r1 := A               -- load
     r2 := B
     r1 := r1 > r2
     r2 := C
     r3 := D
     r2 := r2 > r3
     r1 := r1 & r2
     r2 := E
     r3 := F
     r2 := r2 ≠ r3
     r1 := r1 | r2
     if r1 = 0 goto L2
L1:  then_clause           -- (label not actually used)
     goto L3
L2:  else_clause
L3:
```

SELECTION (using jump code)

- Code generated with short-circuiting (C)

```
            r1 := A
        r2 := B
        if r1 <= r2 goto L4
        r1 := C
        r2 := D
        if r1 > r2 goto L1
        L4:  r1 := E
             r2 := F
             if r1 = r2 goto L2
    L1:  then_clause
             goto L3
        L2:  else_clause
    L3:
```

Selection: Case/switch

- The case/switch statement was introduced in Algol W to simplify certain if-else situations.
- Useful when comparing the same integer to a large variety of possibilities:
  - i := (complex expression)

```
if  i == 1: …
elsif  i in 2,7: …
```

  - Case (complex expression)

```
1: …
2-7: …
```

Using Switch Case

```
CASE ... (* potentially complicated expression *) OF
    1:        clause_A
|   2, 7:     clause_B
|   3..5:     clause_C
|   10:       clause_D
    ELSE      clause_E
END
```

Selection: Case/switch

- While it looks nicer, principle reason is code optimization.
  - Instead of complex branching, just loads possible destinations into simple array.
- Additional implementations:
  - If set of labels is large and sparse (e.g. 1, 2-7, 8-100, 101, 102-105, …) then can make it more space efficient using hash tables or some other data structure

Iteration

- Ability to perform some set of operations repeatedly.
  - Loops
    - Enumeration - Controlled loops
    - Combination loops
    - Logically controlled loops
- In a real sense, this is the most powerful component of programming.
- In general, loops are more common in imperative languages, while recursion is more common in functional languages.

Iteration

- Enumeration-controlled: originated in Fortran
  - Pascal or Fortran-style for loops

```
do i = 1, 10, 2
      …
enddo
```

  - Changed to standard for loops later, eg Modula-2

```
FOR i := first TO last BY step DO
      …
END
```

Iteration: code generation

- At its heart, none of these initial loops allow anything other than enumeration over a preset, fixed number of values.
- This allows very efficient code generation:

```
    R1 := first
    R2 := step
    R3 := last
 L1: …                              --loop body, use
   R1 for i
    R1 := R1 + R2
 L2: if R1 <= R3 goto L1
```

Iteration: code generation

- This can sometimes be optimized if the number of iterations can be precomputed, although need to be careful of overflow.
  - Basically, precompute total iteration count and place it in a register, and decrement at the end of each iteration until we hit 0.
  - Often used in early Fortran compilers.
- Using iteration counts like this avoid the need to test the sign of step within the loop
- This type of prediction is always possible in Fortran or Ada, but C (and its descendants) are quite different.

Iteration: Semantic Complications

- Can control enter or leave the loop other than through enumeration mechanism?
  - Using break/exit statement to leave a *for* loop.
- What happens if the loop body alters variables used to compute end-of-loop bound?
  - Most languages specify that the bound is computed only once, before the first iteration and kept in a temporary location.
- What happens if the loop modifies the index variable itself?
  - Many languages prohibit changes to the loop index within the body of the loop.
- Can the program read the index after the loop has been completed, and if so, what is its value?
  - Depends on scope rules, and is language dependent.

Iteration: Combination Loops

- The for loop in C is called a combination loop - it allows one to use more complex structures in the for loop.
- Essentially, for loops are almost another variant of while loops, with more complex updates and true/false evaluations each time.
- Semantically, the C *for* loop is logically controlled. it was designed to make enumeration easy.

```
for( i=first;  i<=last;  i+=step)
{
......
}
```

Iteration: iterator based loops

- Other languages (Ruby, Python, C# etc.) require any container to provide an iterator that enumerates items in that class.
- This is extremely high level, and relatively new.
- Example: In python,

```
    for i in range(first, last, step):
    .............
```

- Here range is a built –in iterator that yields the integers from first to first+[(last-first)/step] * step in increments of step.
- An iterator is a separate thread of control, with its own program counter, whose execution is interleaved with that of the for loop to which it supplies index values.

Iteration: logically controlled loops

- Here the approach is to test the condition before each iteration.
- Eg) while loop which was introduced in Algol-W.

```
while condition do statement
```

Post-test loops

- Test the terminating condition at the bottom of a loop.
- Eg) In C language,

```
do {
    line = read_line(stdin);
   } while (line[0] != ’$’);
```

Iteration: logically controlled loops

Midtest Loops

- In many languages this "midtest" can be accomplished with a special statement nested inside a conditional: **exit** in Ada, **break** in C, **last** in Perl.
- C uses break to exit the closest for, while, or do loop:

```
for (;;) {
      line = read_line(stdin);
       if (all_blanks(line)) break;
      consume_line(line);
}
```

Recursion

- Recursion
  - equally powerful to iteration
    - **Recursion** is the process of repeating items in a self-similar way.
    - In **programming languages**, if a program allows us to call a function inside the same function, then it is called a **recursive** call of the function.
    - The C **programming language** supports **recursion**, i.e., a function to call itself.

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 based on the repeated modification of variables.
- In recursion, it does not change variables.
- Choosing iteration or recursion is purely based on circumstances.

Iteration and Recursion

- Consider a situation to compute a sum as follows:

$$\sum_{1 \le i \le 10} f(i)$$

- In this case, it is better to use iteration. so the above function can be written in C as,

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
    /* assume low <= high */
    int total = 0;
    int i;
    for (i = low; i <= high; i++) {
        total += f(i);
    }
    return total;
}
```

Iteration and Recursion

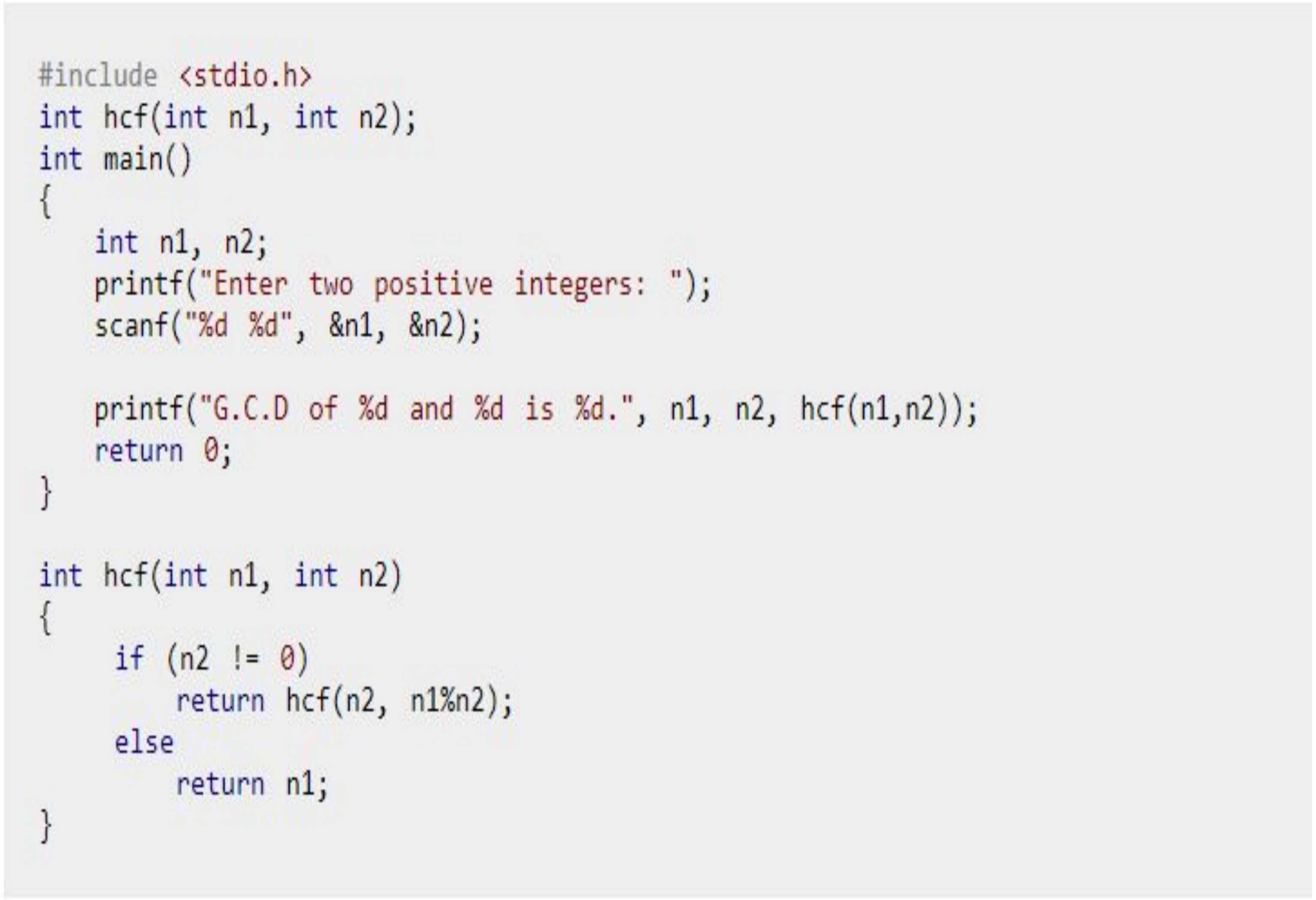
- consider another situation to compute a value defined by a recurrence as,

$$\begin{array}{c} \mathrm{gcd}(a, b) \\ \text{(positive integers, } a, b\text{)} \end{array} \equiv \begin{cases} a & \text{if } a = b \\ \mathrm{gcd}(a-b, b) & \text{if } a > b \\ \mathrm{gcd}(a, b-a) & \text{if } b > a \end{cases}$$

- In this case, it is better to use recursion as follows:

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
    if (a == b) return a;
    else if (a > b) return gcd(a-b, b);
    else return gcd(a, b-a);
}
```

Iteration and Recursion



```c
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);

    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));
    return 0;
}

int hcf(int n1, int n2)
{
     if (n2 != 0)
         return hcf(n2, n1%n2);
     else
         return n1;
}
```

Tail Recursion

- A recursive function is **tail recursive** when recursive call is the last thing executed by the function.
- A tail-recursive function is one in which additional computation never follows a recursive call: the return value is simply whatever the recursive call returns.
- For such functions, dynamically allocated stack space is unnecessary: the compiler can reuse the space belonging to the current iteration when it makes the recursive call.

Tail Recursion

- The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler.
- The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use.

```
void print(int n)
{
    if (n < 0)  return;
    cout << " " << n;
    // The last executed statement is recursive call
   print(n-1);
}
```

Applicative- and Normal-Order Evaluation

- when arguments are evaluated before passing them to a subroutine, then it is known as **applicative-order evaluation.**
- When arguments are evaluated only when the value is actually needed, then it is known as **normal order evaluation**. ie, It is possible to pass a representation of the unevaluated arguments to the subroutine.
- Normal-order evaluation is what naturally occurs in macros.
- It also occurs in short-circuit Boolean evaluation, call-by-reference parameters and certain functional languages.
- Scheme is an applicative-order language, namely, that all the arguments to Scheme procedures are evaluated when the procedure is applied.

Applicative- and Normal-Order Evaluation

- Applicative order is a call-by-value evaluation.
- Normal order is a call-by-reference evaluation.
- Consider an example in Scheme language.

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

- run the following procedure call:

```
(square (square 4))
```

How many times is * called?

The answer to this question differs depending on whether we are using applicative or normal order.

Applicative- and Normal-Order Evaluation

- In applicative, we evaluate the arguments first:

(square (square 4))

|____| |_________|

procedure *argument*

- So we need to first find the value of the inner (square 4) before we can evaluate the outer one. The same process needs to be repeated:

(square (square 4))

|____| |

procedure *argument*

- This time, the argument is self-evaluating, so we don't need to evaluate the argument. We can now pass this evaluated argument into square, producing the following equivalent expression:

Applicative- and Normal-Order Evaluation

(square (* 4 4))

|____| |______|

procedure   *argument*

- Once again, we evaluate the argument:

  (square 16)   // *At this point, * has been called once*

- And now we can pass this evaluated argument in once again:

  (* 16 16) // *After this expression is evaluated, * will have been called twice.*

- This expression gives us our final answer of 256, with two calls to *.

Applicative- and Normal-Order Evaluation

- In normal order, we delay our argument evaluation until we actually need its value:

```
 (square (square 4))
  |____|  |________|
 procedure  argument
```

- So in this case, we pass the unevaluated argument directly into the procedure:

```
 (* (square 4) (square 4))
   | |______________________|
proced.          arguments
```

- Since we're working in normal order, we'd like to be able to pass the arguments directly into *.

Applicative- and Normal-Order Evaluation

- However, multiplying procedure calls together doesn't make sense, so we can't delay argument evaluation any further, and are forced to evaluate (square 4):

  (* (* 4 4) (* 4 4))

- At this point it's fairly clear how many calls to * there will be, but we'll continue with this process anyway. We evaluate each of the arguments:

  (* 16 16) // *At this point, two calls to * have been made*

  // *One more call will be made to evaluate this expression*

Now we can finally pass those arguments into * to get our final answer of 256, with three calls to *.

Lazy Evaluation

- It is an evaluation strategy which delays the evaluation of an expression until its value is needed.
- So it is called as **call-by-need.**
- its like a normal-order evaluation that will sometimes not evaluate an argument at all, if its value is never actually needed.
- 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.

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.**
- A common use of lazy evaluation is to create so-called *infinite* or *lazy data structures* that are“fleshedout”on demand.

Lazy Evaluation

- The **range** method in Python follows the concept of Lazy Evaluation. It avoids repeated evaluation.
- It saves the execution time for larger ranges and we never require all the values at a time, so it saves memory consumption as well.
- Consider the following example.

```
r = range(10)
print(r)
range(0, 10)
print(r[3])
```

- It will produce the following output −

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3
```

Nondeterminacy

- A nondeterministic construct is one in which the choice between alternatives(i.e.,between control paths) is deliberately unspecified.
- We have already seen examples of nondeterminacy in the evaluation of expressions in most languages, operator or subroutine arguments may be evaluated in any order.
- Some languages, notably Algol68 and various concurrent languages, provide more extensive nondeterministic mechanisms, which cover statements as well