SCHEMES AND SYLLABUS

PROGRAMMING LANGUAGES

Subject Code: 10CS666 I.A. Marks : 25
Hours/Week : 04 Exam Hours: 03
Total Hours : 52 Exam Marks: 100

PART - A

UNIT – 1 7 Hours
Introduction; Names, Scopes, and Bindings: The art of language design; Programming language spectrum; Why study programming languages? Compilation and interpretation; Programming environments. Names, scope, and bindings: The notion of binding time; Object lifetime and storage management; Scope rules; Implementing scope; The meaning of names within a scope; The binding of referencing environments; Macro expansion.

UNIT – 2 7 Hours
Control Flow: Expression evaluation; Structured and unstructured flow Sequencing; Selection; Iteration; Recursion; Non-determinacy

UNIT – 3 6 Hours
Data Types: Type systems; Type checking; Records and variants; Arrays;Strings; Sets; Pointers and recursive types; Lists; Files and Input/Output; Equality testing and assignment.

UNIT – 4 6 Hours
Subroutines and Control Abstraction: Review of stack layout; Calling sequences; Parameter passing; Generic subroutines and modules; Exception handling; Coroutines; Events.

PART – B

UNIT – 5 6 Hours
Data Abstraction and Object Orientation: Object oriented programming; Encapsulation and Inheritance; Initialization and finalization; Dynamic method binding; Multiple inheritance; Object oriented programming revisited.

UNIT – 6 7 Hours
Functional Languages, and Logic Languages: Functional Languages: Origins; Concepts; A review/overview of scheme; Evaluation order revisited; Higher-order functions; Functional programming in perspective. Logic Languages: Concepts; Prolog; Logic programming in perspective.

UNIT – 7 6 Hours
Concurrency: Background and motivation; Concurrency programming fundamentals; Implementing synchronization; Language-level mechanisms; Message passing.

UNIT – 8 7 Hours

Run-Time Program Management: Virtual machines; Late binding of machine code; Inspection/introspection.

Text Books:

1. Michael L. Scott: Programming Language Pragmatics, 3Edition, Elsevier, 2009.
(Chapters 1.1 to 1.5, 3.1 to 3.7, 6 excluding the sections on CD, 7 excluding the ML type system, 8, 9, 10 excluding the sections on CD, 11 excluding the sections on CD, 12, 15. Note: Text Boxestitled Design & Implementation are excluded)

Reference Books:

1. Ravi Sethi: Programming languages Concepts and Constructs, 2 Edition, Pearson Education, 1996.
2. R Sebesta: Concepts of Programming Languages, 8 Edition, Pearson Education, 2008.
3. Allen Tucker, Robert Nonan: Programming Languages, Principles Paradigms, 2nd Edition, Tata McGraw-Hill, 2007.

TABLE OF CONTENT

1.	Introduction; Names, Scopes, and Bindings	5-12
2.	Control Flow	14-20
3.	Data Types	22-38
4.	Subroutines and Control Abstraction	40-60
5.	Data Abstraction and Object Orientation	62-67
6.	Functional Languages, and Logic Languages	69-80
7.	Concurrency	82-93
8.	Run-Time Program Management	95-96

UNIT – 1
Introduction; Names, Scopes, and Bindings:

- The art of language design
- Programming language spectrum
- Why study programming languages?
- Compilation and interpretation
- Programming environments
- Names, scope, and bindings
- The notion of binding time
- Object lifetime and storage management
- Scope rules
- Implementing scope
- The meaning of names within a scope
- The binding of referencing environments
- Macro expansion.

UNIT 1

Introduction:

- Assembly languages were originally designed with a one-to-one correspondence between mnemonics and machine language instructions. Translating from mnemonics to machine language became the job of a systems program known as an *assembler*.

- Assemblers were eventually augmented with elaborate "macro expansion" facilities to permit programmers to define parameterized abbreviations for common sequences of instructions.

- As computers evolved, and as competing designs developed, it became increasingly frustrating to have to rewrite programs for every new machine. It also became increasingly difficult for human beings to keep track of the wealth of detail in large assembly language programs.

- Translating from a high-level language to assembly or machine language is the job of a systems program known as a *compiler*. Compilers are substantially more complicated than assemblers because the one-to-one correspondence between source and target operations no longer exists when the source is a high-level language.

The Art of Language Design:

Why there are so many programming languages:

Evolution: Computer science is a young discipline; we're constantly finding better ways to do things. The late 1960s and early 1970s saw a revolution in "structured programming," in which the go to-based control flow of languages like Fortran, Cobol, and Basic2 gave way to while loops, case statements, and similar higher-level constructs.

.*Special Purposes:* Many languages were designed for a specific problem domain. The various Lisp dialects are good for manipulating symbolic data and complex data structures. . C is good for low-level systems rogramming. Each of these languages can be used successfully for a wider range of tasks, but the emphasis is clearly on the specialty.

Personal Preference: Different people like different things. Some people find it natural to think recursively; others prefer iteration. Some people like to work with pointers; others prefer the implicit dereferencing of Lisp, Clu, Java, and ML. The strength and variety of personal preference make it unlikely that anyone will ever develop a universally acceptable programming language.

Expressive Power: One commonly hears arguments that one language is more "powerful" than another,though in a formal mathematical sense they are all Turing equivalent. The factors that contribute to expressive power—abstraction facilities in particular

Open Source: Most programming languages today have at least one open source compiler or interpreter, but some languages—C in particular—are much more closely associated than others with freely distributed, peer reviewed, community supported computing.

Why Study Programming Languages:

Understand obscure features.: The typical C++ programmer rarely uses unions,multiple inheritance, variable numbers of arguments, or the .* operator. Just as it simplifies the assimilation of new languages, an understanding of basic concepts makes it easierto understand these features when you look up the details in the manual.

Choose among alternative ways to express things: based on a knowledge of implementation costs. In C++, for example, programmers may need to avoid unnecessary temporary variables, and use copy constructors whenever possible, to minimize the cost of initialization..

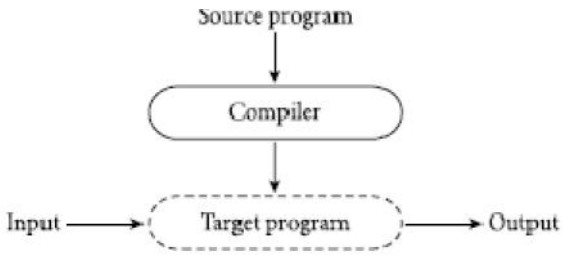
Make good use of debuggers, assemblers, linkers, and related tool:. In general, the high-level language programmer should not need to bother with implementation details. There are times, however, when an understanding of those details proves extremely useful.

Simulate useful features in languages that lack them: Certain very useful features are missing in older languages but can be emulated by following a deliberate programming style. In older dialects of Fortran, for example, programmers familiar with modern control constructs can use comments and self-discipline to write well-structured code. In Fortran 77 and other languages that lack recursion, an iterative program can be derived via mechanical hand transformations, starting with recursive pseudocode.

Make better use of language technology wherever it appears: Most programmers will never design or implement a conventional programming language, but most will need language technology for other programming tasks. The typical personal computer contains files in dozens of structured formats, encompassing web content, word processing, spreadsheets, presentations, raster and vector graphics, music, video, databases, and a wide variety of other application domains.

Compilation and Interpretation

At the highest level of abstraction, in a high-level language look something like this:



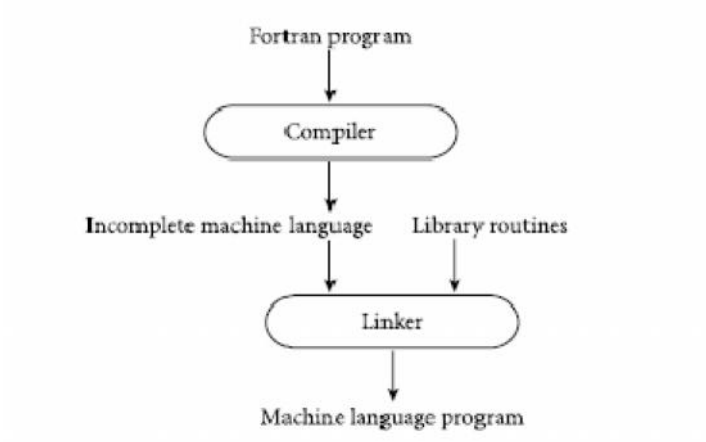
- The compiler *translates* the high-level source program into an equivalent target program (typically in machine language) and then goes away. At some arbitrary later time, the user tells the operating system to run the target program.

- The compiler is the locus of control during compilation; the target program is the locus of control during its own execution. The compiler is itself a machine language program, presumably created by compiling some other high-level program. When written to a file in a format understood by the operating system, machine language is commonly known as *object code*.

- An alternative style of implementation for high-level languages is known as *interpretation*.an interpreter stays around for the execution of the application. In fact, the interpreter is the locus of control during that execution. The interpreter implements a virtual machine whose "machine language" is the high-level programming language.

- The interpreter reads statements in that language more or less one at a time, executing them as it goes along.

- The translates Fortran source into machine language.it counts on the existence of a *library* of subroutines that are not part of the original program. Examples include mathematical functions (sin, cos, log, etc.) and I/O. The compiler relies on a separate program, known as a *linker*, to merge the appropriate library routines into the final program:



Programming Environments

- In older programming environments, tools may be executed individually, at the explicit request of the user. If a running program terminates abnormally with a "bus error"

(invalid address) message, for example, the user may choose to invoke a debugger to examine the "core" file dumped by the operating system.

- He or she may then attempt to identify the program bug by setting breakpoints, enabling tracing, and so on, and running the program again under the control of the debugger.
- Once the bug is found, the user will invoke the editor to make an appropriate change. He or she will then recompile the modified program, possibly with the help of a configuration manager.
- More recent programming environments provide much more integrated tools. When an invalid address error occurs in an integrated environment, a new window is likely to appear on the user's screen, with the line of source code at which the error occurred highlighted.
- Breakpoints and tracing can then be set in this window without explicitly invoking a debugger. Changes to the source canbe made without explicitly invoking an editor.
- The editor may also incorporate knowledge of the language syntax, providing templates for all the standard control structures, and checking syntax as it is typed in.
- If the user asks to rerun the program after making changes, a new version may be built without explicitly invoking the compiler or configuration manager.

Concept of binding time

A *binding* is an association between two things, such as a name and the thing itnames. *Binding time* is the time at which a binding is created or, more generally, the time at which any implementation decision is made.

Situations where binding time exists:

Language design time: In most languages, the control flow constructs, the set of fundamental (primitive) types, the available *constructors* for creating complex types, and many other aspects of language semantics are chosen when the language is designed.

Language implementation time: Most language manuals leave a variety of issues to the discretion of the language implementor. Typical examples include the precision (number of bits) of the fundamental types, the coupling of I/O to the operating system's notion of files, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow

Program writing time: Programmers, choose algorithms, data structures, and names.

Compile time: Compilers choose the mapping of high-level constructs to machine code, including the layout of statically defined data in memory.

Link time: Since most compilers support *separate compilation*—compiling different modules of a program at different times—and depend on the availability of a library of standard subroutines, a program is usually not complete until the various modules are joined together by a linker. The

linker chooses the overall layout of the modules with respect to one another. It also resolves intermodule references..

Load time: Load time refers to the point at which the operating system loads the program into memory so that it can run. In primitive operating systems, the choice of machine addresses for objects within the program was not finalized until load time.

Object lifetime and storage management:

- The period of time between the creation and the destruction of a name-toobject binding is called the binding's *lifetime*. Similarly, the time between the creation and destruction of an object is the object's lifetime.
- These lifetimes need not necessarily coincide. In particular, an object may retain its value and the potential to be accessed even when a given name can no longer be used to access
- Object lifetimes generally correspond to one of three principal *storage allocation* mechanisms, used to manage the object's space:

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

2. *Stack* objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.

3. *Heap* objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm.

Static Allocation

- The instructions that constitute a program's machine-language translation can also be thought of as statically allocated objects.
- Numeric and string-valued constant literals are also statically allocated, for statements such as A = B/14.7 or printf("hello, world\n")
- Statically allocated objects whose value should not change during program execution (e.g., instructions, constants, and certain run-time tables) are often allocated in protected, read-only memory so that any inadvertent attempt to write to them will cause a processor interrupt, allowing the operating system to announce a run-time error.

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

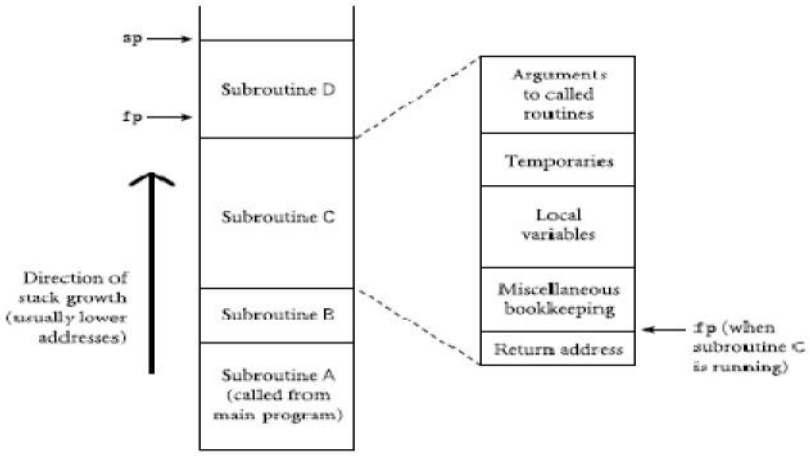
Arguments and return values. Modern compilers tend to keep these in registers when possible, but sometimes space in memory is needed.

Temporaries. These are usually intermediate values produced in complex calculations. Again, 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 additional saved registers, debugging information

Stack-Based Allocation

- If a language permits recursion, static allocation of local variables is no longer an option, since the number of instances of a variable that may need to exist at the same time is conceptually unbounded.
- Each instance of a subroutine at run time has its own *frame* (also called an *activation record*) on the stack, containing arguments and return values, local variables, temporaries, and bookkeeping



- Maintenance of the stack is the responsibility of the subroutine *calling sequence* the code executed by the caller immediately before and after the call and of the *prologue* (code executed at the beginning) and *epilogue* (code executed at the end) of the subroutine itself.
- Sometimes the term"calling sequence" is used to refer to the combined operations of the caller, the prologue, and the epilogue. While the location of a stack frame cannot be predicted at compile time the offsets of objects *within* a frame usually *can* be statically determined.
- Code that needs to access a local variable within the current frame, or an argument near the top of the calling frame, can do so by adding a predetermined offset to the value in the frame pointer.

Garbage Collection

- Allocation of heap-based objects is always triggered by some specific operation in a program: instantiating an object, appending to the end of a list, assigning a long value into a previously short string, and so on.

- The run-time library for such a language must then provide a *garbage collection* mechanism to identify and reclaim unreachable objects. If the programmer can correctly identify the end of an object's lifetime, without too much run-time the result is likely to be faster execution.

- If an object is deallocated too soon, the program may follow a *dangling reference*, accessing memory now used by another object.If an object is *not* deallocated at the end of its lifetime, then the program may "leak memory," eventually running out of heap space. Deallocation errors are difficult to identify and fix.

Scope Rules:

- In most modern languages, the scope of a binding is determined statically—that is, at compile time. In C, for example, we introduce a new scope upon entry to a subroutine.

- We create bindings for local objects and deactivate bindings for global objects that are "hidden" by local objects of the same name. On subroutine exit, we destroy bindings for local variables and reactivate bindings for any global objects that were hidden

- A scope is the body of a module, class, subroutine, or structured control flow statement, sometimes called a *block*. In C family languages it would be delimited with {...} braces.

Static Scope

- In a language with static (lexical) scoping, the bindings between names and objects can be determined at compile time by examining the text of the program, without consideration of the flow of control at run time.

- The scope of a local variable is limited to the subroutine in which it appears; it is not visible elsewhere. Variable declarations are optional. 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.

- the lifetime of a local Fortran variable encompasses a single execution of the variable's subroutine. Programmers can override this rule by using an explicit save statement

- A save-ed variable has a lifetime that encompasses the entire execution of the program In a Fortran compiler that uses a stack to save space, or that exploits knowledge of the patterns of calls among subroutines to overlap statically allocated space

Nested Blocks

- local variables can be declared not only at the beginning of any subroutine, but also at the top of any begin. . . end ({...}) block. Others languages, including Algol 68, C99, and all

of C's descendants, are even more flexible, allowing declarations wherever a statement may appear.

- Variables declared in nested blocks can be very useful, as for example in the inner declarations in C following C code.

```
{
int temp = a;
a = b;
b = temp;
}
```

- Keeping the declaration of temp lexically adjacent to the code that uses it makes the program easier to read, and eliminates any possibility that this code will interfere with another variable named temp.

- No run-time work is needed to allocate or deallocate space for variables declared in nested blocks; their space can be included in the total space for local variables allocated in the subroutine prologue and deallocated in the epilogue.

UNIT – 2

Control Flow

- Expression evaluation
- Structured and unstructured flow
- Sequencing
- Selection
- Iteration
- Recursion
- Non-determinacy

UNIT 2

1. Structured and Unstructured Flow:

- Early versions of Fortran approach by relying heavily on goto statements for most nonprocedural control flow:

```
if A .lt. B goto 10 ! ".lt." means "<"
...
10
```

The 10 on the bottom line is a *statement label*.

- The abandonment of gotos was part of a larger "revolution" in software engineering known as *structured programming*.

- Structured programming emphasizes top-down design modularization of code, structured types descriptive variable and constant names, and extensive commenting conventions.

Structured Alternatives to goto

Mid-loop exit and continue: A common use of gotos in Pascal was to break out Leaving the middle of a loop:

```
while not eof do begin
readln(line);
if all_blanks(line) then goto 100;
consume_line(line)
end;
100:
```

Multilevel returns: Returns and (local) gotos allow control to return from the current subroutine. On occasion it may make sense to return from a routine.

```
function search(key : string) : string;
var rtn : string;
...
procedure search_file(fname : string);
...
begin
...
for ... (* iterate over lines *)
...
if found(key, line) then begin
rtn := line;
goto 100;
end;
...
```

end;

Continuations

- The notion of nonlocal gotos that unwind the stack can be generalized by defining what are known as *continuations*.
- 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.
- *c* can be saved in variables, returned explicitly by subroutines, or called repeatedly, even after control has returned from *f*

2. Sequencing:

- It is the means of controlling the order in which side effects occur, when one statement follows another in the program text, the first statement executes before the second.
- lists of statements can be enclosed with begin. . . end or {. . . } delimiters and then used in any context in which a single statement is expected.

Ex:
procedure srand(seed : integer)
— Initialize internal tables.
— The pseudo-random generator will return a different
-sequence of values for each different value of seed.

function rand() : integer

-No arguments; returns a new “random” number.

rand needs to have a side effect, so that it will return a different value each time it is called.

3. Selection:

Selection statements in most imperative languages implements variant of the
Selection in Algol 60 if. . . then . . . else notation introduced in Algol 60:

if *condition* then *statement*
else if *condition* then *statement*
else if *condition* then *statement*
...
else *statement* _

most languages with terminators provide a special elsif or elif keyword. In Modula-2, one writes

IF a = b THEN ...
ELSIF a = c THEN ...
ELSIF a = d THEN ...
ELSE ...
END

Short-Circuited Conditions

- While the condition in an if. . . then . . . else statement is a Boolean expression, there is usually no need for evaluation of that expression to result in a Boolean value in a register.

- Most machines provide conditional branch instructions that capture simple comparisons. 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.

Ex: Code generation for a Boolean condition

if ((A > B) and (C > D)) or (E _= F) then
then clause
else
else clause

Case/Switch Statements :

The case statements of Algol provide alternative syntax for a special case of nested if. . . then . . . else

i := ... (* potentially complicated expression *)
IF i = 1 THEN
clause A
ELSIF i IN 2, 7 THEN
clause B
ELSIF i IN 3..5 THEN
clause C
ELSIF (i = 10) THEN
clause D
ELSE
clause E
END

can be rewritten as

CASE ... (* potentially complicated expression *) OF
1: *clause A*
| 2, 7: *clause B*
| 3..5: *clause C*
| 10: *clause D*
ELSE *clause E*
END

code fragments (*clause A*, *clause B*, etc.) after the colons and the ELSE are called the *arms* of the CASE statement. The lists of constants in front of the colons are CASE statement *labels*.

The C* switch *Statement:

C's syntax for case (switch) statements
.
switch (... /* tested expression */) {
case 1: *clause A*
break;
case 2:
case 7: *clause B*
break;
case 3:
case 4:
case 5: *clause C*
break;
case 10: *clause D*
break;
default: *clause E*
break;
}

- Each possible value for the tested expression must have its own label within the switch; ranges are not allowed. Lists of labels are not allowed, but the effect of lists can be achieved by allowing a label to have an *empty* arm that simply "falls through" into the code for the subsequent label.

- An explicit break statement must be used to get out of the switch at the end of an arm, rather than falling through into the next.

4. <u>Iteration:</u>

- Iteration and recursion are the two mechanisms that allow a computer to perform similar operations repeatedly. Without at least one of these mechanisms, the running time of a

program is a linear function of the size of the program text, and the computational power of the language is no greater than that of a finite automaton.

- In most languages, iteration takes the form of *loops*. Like the statements in a sequence, the iterations of a loop are generally executed for their side effects: their modifications of variables.

- An *enumeration-controlled* loop is executed once for every value in a given finite set. The number of iterations is therefore known before the first iteration begins.

- A *logically controlled* loop is executed until some Boolean condition changes value.

Enumeration-Controlled Loops:

```
do 10 i = 1, 10, 2
...
10 continue
```

- The number after the do is a label that must appear on some statement later in the current subroutine; the statement it labels is the last one in the *body* of the loop: the code that is to be executed multiple times.

- Continue is a "no-op": a statement that has no effect. Using a continue for the final statement of the loop makes it easier to modify code

Serious problems with the Fortran do loop are a :

- If statements in the body of the loop change the value of i, then the loop may execute a different number of times than one would assume based on the bounds in its header.

- code that simply jumps in, without properly initializing i,almost certainly represents a programming error, but will not be caught by the compiler.

They must be addressed in the design of enumeration-controlled loops in any language.

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

where first, last, and step can be arbitrarily complex expressions of an integer, enumeration, or subrange type.

Combination Loops

- Algol 60, provides a single loop construct that subsumes. Algol 60 for loop the properties of more modern enumeration- and logically controlled loops.

The general form is given by
for stmt −→ for *id* := *for list* do *stmt*
for list −→ *enumerator* **(** , *enumerator* **)***
enumerator −→ *expr*
−→ *expr* step *expr* until *expr*
−→ *expr* while *condition*

- Here the index variable takes on values specified by a sequence of enumerators, each of which can be a single value, a range of values similar to that of modern enumeration-controlled loops, or an expression with a terminating condition.

5. Recursion:

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

To compute a value defined by a recurrence, A "naturally recursive" problem

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

recursion may seem more natural:

```
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);
}
```

Fibonacci numbers, are defined by the mathematical recurrence:

$$\underset{\text{(nonnegative integer } n)}{F_n} \equiv \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

6. Non determinacy:

- 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, Algol 68 and various concurrent languages, provide more extensive nondeterministic mechanisms, which cover statements as well.

UNIT – 3

Data Types

- Type systems
- Type checking
- Records and variants
- Arrays
- Strings
- Sets
- Pointers and recursive types
- Lists
- Files and Input/Output
- Equality testing and assignment.

UNIT 3

1. Type systems :

- Types provide implicit context for many operations, so the programmer does not have to specify that context explicitly.
- In Pascal, for instance, the expression a + b will use integer addition if a and b are of integer type; it will use floating-point addition if a and b are of real type.
- Types limit the set of operations that may be performed in a semantically Errors captured by type information valid program. They prevent the programmer from adding a character and a record
- A *type system* consists of (1) a mechanism to define types and associate them with certain language constructs and (2) a set of rules for *type equivalence, type compatibility*, and *type inference*.

Type equivalence rules determine when the types of two values are the same. Type compatibility rules determine when a value of a given type can be used in a given on text.

Type inference rules define the type of an expression based on the types of its constituent parts the surrounding context.

Type Checking

- *Type checking* is the process of ensuring that a program obeys the language's type compatibility rules. A violation of the rules is known as a *type clash*.
- A violation of the rules is known as a *type clash*. A language is said to be *strongly typed* if it prohibits, in a way that the language implementation can enforce, the application of any operation to any object that is not intended to support that operation
- language is said to be *statically typed* if it is strongly typed and type checking can be performed at compile time.
- Ex: A Pascal implementation can also do most of its type checking at compile time, though the language is not quite strongly typed: untagged variant records are its only loophole.

Polymorphism

- *Polymorphism* allows a single body of code to work with objects of multiple types. It may or may not imply the need for run-time type checking.

- Only at run time does the language implementation check to see that the objects actually implement the requested operations. Because the types of objects can be thought of as implied (unspecified) parameters, dynamic typing is said to support *implicit parametric polymorphism*.

- In object-oriented languages, *subtype polymorphism* allows a variable *X* of type *T* to refer to an object of any type derived from *T*. The compiler can be sure that any operation acceptable for an object of type *T* will be acceptable for any object referred to by *X*- *explicit parametric polymorphism*

The Definition of Types

- There are at least three ways to think about types, which we may call the *denotational*, *constructive*, and *abstraction-based* points of view.

- From the denotational point of view, a type is simply a set of values. A value has a given type if it belongs to the set; an object has a given type if its value is guaranteed to be in the set.

- From the constructive point of view, a type is either one of a small collection of *builtin* types or a *composite* type created by applying a type *constructor* (record, array, set, etc.) to one or more simpler type.

Numeric Types

- A few languages (e.g., C and Fortran) distinguish between different lengths of integers and real numbers; most do not, and leave the choice of precision to the implementation.

- A few languages, including C, C++, C#, and Modula-2, provide both signed and unsigned integers.

- Ada supports *fixed-point* types, which are represented internally by integers but have an implied decimal point at a programmer-specified position among the digits. Fixed-point numbers provide a compact representation of non-integral values (e.g., dollars and cents) within a restricted range.

Enumeration Types

- They facilitate the creation of readable programs, and allow the compiler to catch certain kinds of programming errors. An enumeration type consists of a set of named elements.

- In Pascal, one can write Enumerations in Pascal

    - type weekday = (sun, mon, tue, wed, thu, fri, sat);

- An alternative to enumerations, of course, is simply to declare a collection of enumerations as constants:
    - const sun = 0; mon = 1; tue = 2; wed = 3; thu = 4; fri = 5; sat = 6;
- In C, the difference between the two approaches is purely syntactic. The declaration
    - enum weekday {sun, mon, tue, wed, thu, fri, sat};
- Values of an enumeration type are typically represented by small integers, usually a consecutive range of small integers starting at zero. In many languages these *ordinal values* are semantically significant, because built-in functions can be used to convert an enumeration value to its ordinal value

Composite Types

Nonscalar types are usually called *composite*, or *constructed* types. They are generally created by applying a *type constructor* to one or more simpler types.

Common composite types include records (structures), variant records (unions), arrays, sets, pointers, lists, and files.

Records: A record consists of a collection of *fields*, each of which belongs to a simpler type.

Variant records: differ from "normal" records in that only one of a variant record's fields (or collections of fields) is valid at any given time. A variant record type is the *union* of its field types, rather than their Cartesian product.

Arrays : are the most commonly used composite types. An array can be thought of as a function that maps members of an *index* type to members of a *component* type.

Arrays of characters are often referred to as *strings*, and are often supported by special purpose operations not available for other arrays.

Sets: like enumerations and subranges, were introduced by Pascal. A set type is the mathematical powerset of its base type, which must usually be discrete.

A variable of a set type contains a collection of distinct elements of the base type.

Pointers are l-values. A pointer value is a *reference* to an object of the pointer's base type. Pointers are often but not always implemented as addresses. They are most often used to implement *recursive* data types.

A type *T* is recursive if an object of type *T* may contain one or more references to other objects of
type *T*.

2. Type Checking:

Whenever an expression is constructed from simpler sub expressions using different ways:

Type Equivalence

There are two ways of defining type equivalence.

- *Structural equivalence* is based on the *content* of type definitions: roughly speaking, two types are the same if they consist of the same components, put together in the same way.

- *Name equivalence* is based on the *lexical occurrence* of type definitions: roughly speaking, each definition introduces a new type.

- Its principal problem of *Structural equivalence* is an inability to distinguish between types that the programmer may think of as distinct, but which happen by coincidence to have the same internal structure:

- Name equivalence is based on the assumption that if the programmer takes the effort to write two type definitions, then those definitions are probably meant to represent different types.

- A language in which aliased types are considered distinct is said to have *strict name equivalence*. A language in which aliased types are considered equivalent is said to have *loose name equivalence*.

- One way to think about the difference between strict and loose name equivalence is to remember the distinction between declarations and definitions

- Under strict name equivalence, a declaration type A = B is considered a definition. Under loose name equivalence it is merely a declaration; A shares the definition of B.

Type Conversion and Casts

- if the programmer wishes to use a value of one type in a context that expects another, he or she will need to specify an explicit *type conversion* (also sometimes called a type *cast*).

- Depending on the types involved, the conversion may ormay not require code to be executed at run time.

There are three principal cases:

- The types would be considered structurally equivalent, but the language uses name equivalence. In this case the types employ the same low-level representation, and have the same set of values

- The types have different sets of values, but the intersecting values are represented in the same way. One type may be a subrange of the other, for example, or one may consist of two's complement signed integers, while the other is unsigned.

- The types have different low-level representations, but we can nonetheless define some sort of correspondence among their values. A 32-bit integer, for example, can be converted to a double-precision IEEE floating-point number with no loss of precision.

A type conversion in C is specified by using the name of the desired type, in parentheses, as a prefix operator:

```
r = (float) n; /* generates code for run-time conversion */
n = (int) r; /* also run-time conversion, with no overflow check */
```

C and its descendants do not by default perform run-time checks for arithmetic overflow on any operation, though such checks can be enabled if desired in C#.

Generic Reference Types

- To facilitate the writing of general purpose *container* (*collection*) objects (lists, stacks, queues, sets, etc.) that hold references to other objects, several languages provide a "generic reference" type.

- In C and C++, this type is called void * Arbitrary l-values can be assigned into an object of generic reference type, with no concern about type safety: because the type of the object referred to by a generic reference is unknown ,the compiler will not allow any operations to be performed on that object.
- One way to ensure the safety of generic to specific assignments is to make objects self-descriptive—that is, to include in the representation of each object an indication of its type.

- C++ minimizes the overhead of type tags by permitting dynamic_cast operations only on objects of polymorphic types

3. Records (Structures) and Variants (Unions)

Record types allow related data of heterogeneous types to be stored andmanipulated together. Some languages (notably Algol 68, C, C++, and Common Lisp) use the term *structure* (declared with the keyword struct) instead of *record*.

Structures in C++ are defined as a special form of *class* (one in which members are globally visible by default). Java has no distinguished notion of struct; its programmers use classes in all cases

Syntax and Operations

In C, the corresponding declaration would be A C struct

```
struct element {
char name[2];
int atomic_number;
double atomic_weight;
_Bool metallic;
};
```

Accessing members:

Each of the record components is known as a *field*. To refer to a given field of a record, most languages use "dot" notation.

```
type short_string = packed array [1..30] of char;
type ore = record
name : short_string;
element_yielded : record
name : two_chars;
atomic_number : integer;
atomic_weight : real;
metallic : Boolean
end
end;
```

Memory Layout and Its Impact

- The fields of a record are usually stored in adjacent locations in memory. In its symbol table, the compiler keeps track of the offset of each field within each record type.
- When it needs to access a field, the compiler typically generates a load or store instruction with displacement addressing
- For a local object, the base register is the frame pointer; for a global object, the base register is the global pointer. In either case, the displacement is the sum of the record's offset from the register and the field's offset within the record.
- layout for our element type on a 32-bit machine appears in Figure . Because the name field is only two characters long, it occupies two bytes in memory.
- Since atomic_number is an integer, and must (on most machines) be longword-aligned, there is a two-byte "hole" between the end of name and the be ginning of atomic_number.

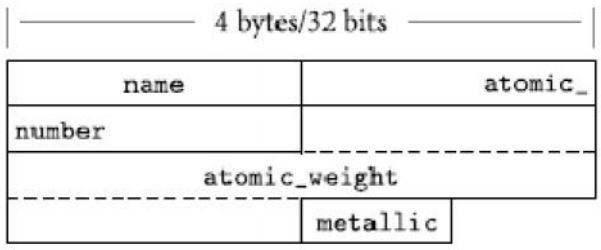


Fig: Memory layout for a record type

- For small records, both copies and comparisons can be performed in-line on a field-by-field basis. For longer records, we can save significantly on code space by deferring to a library routine.

- One solution is to arrange for all holes to contain some predictable value (e.g., zero), but this requires code at every elaboration point. Another is to have the compiler generate a customized field-by-field comparison routine for every record type.

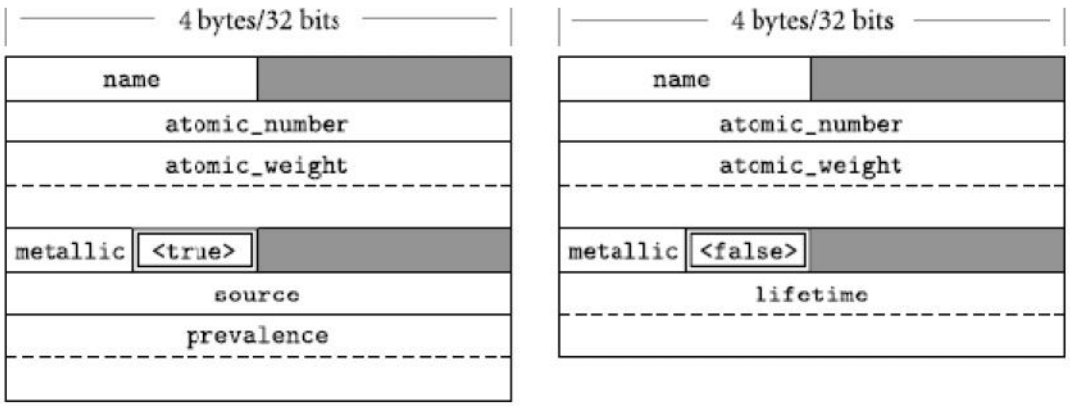
Variant Records

variant record provides two or more alternative fields or collections of fields, only one of which is valid at any given time.

Variant records have their roots in the equivalence statement of Fortran I and in the union types of Algol 68. The Fortran syntax looks like this:

```
Fortran equivalence
statement integer i
real r
logical b
equivalence (i, r, b)
```

The equivalence statement informs the compiler that i, r, and b will never be used at the same time, and should share the same space in memory.



Likely memory layouts for element variants

In c, union looks like

```
struct element {
char name[2];
int atomic_number;
double atomic_weight;
_Bool metallic;
_Bool naturally_occurring;
union {
struct {
char *source;
double prevalence;
} natural_info;
double lifetime;
} extra_fields;
} copper;
```

- Because the union is not a part of the struct, we have to introduce two extra levels of naming. The third field is still copper.atomic_weight, but the source field must be accessed as copper.extra_fields.natural_info.source.

4. Arrays

- Arrays are the most common and important composite data types. They have been a fundamental part of almost every high-level language.
- Unlike records, which group related fields of disparate types, arrays are usually homogeneous. Semantically, they can be thought of as a mapping from an *index type* to a *component* or *element type*.

Syntax and Operations

- Most languages refer to an element of an array by appending a subscript delimited by parentheses or square brackets—to the name of the array. In Fortran and Ada, one says A(3); in Pascal and C, one says A[3].

- Since parentheses are generally used to delimit the arguments to a subroutine call, square bracket subscript notation has the advantage of distinguishing between the two.

Declarations

some languages declares an array by appending subscript notation to the Array declarations syntax that would be used to declare a scalar.

In C:

```
char upper[26];
```

In Fortran:

```
character, dimension (1:26) :: upper
character (26) upper ! shorthand notation
```

In C, the lower bound of an index range is always zero: the indices of an *n*-element array are 0 . . *n* − 1. In Fortran, the lower bound of the index range is one by default.

Multi-dimensional arrays

Most languages make it easy to declare multidimensional arrays:

Multidimensional arrays
matrix : array (1..10, 1..10) of real; -- Ada
real, dimension (10,10) :: matrix
In Ada, by contrast,
Multidimensional v. built-up
arrays matrix : array (1..10, 1..10) of real;
is not the same as
matrix : array (1..10) of array (1..10) of real;

The former is a two-dimensional array, while the latter is an array of one dimensional arrays. With the former declaration, we can access individual real numbers as matrix(3,4);

Arrays of arrays in C

In C, one must also declare, and use two-subscript notation, but C's integration of pointers and arrays means that slices are not supported.

```
double matrix[10][10];
```

Given this definition, matrix[3][4] denotes an individual element of the array, but matrix[3] denotes a *reference*, either to the third row of the array or to the first element of that row, depending on context.

Slices and Array Operations

- A *slice* or *section* is a rectangular portion of an array. Ada provides more limited support: a slice is simply a contiguous range of elements in a one-dimensional array.

- Ada allows one-dimensional arrays whose elements are discrete to be compared for *lexicographic ordering*: A < B if the first element of A that is not equal to the corresponding element of B is less than that corresponding element.

- Fortran 90 has a very rich set of *array operations*: built-in operations that take entire arrays as arguments.

- Any of the built-in arithmetic operators will take arrays as operands; the result is an array,of the same shape as the operands, whose elements are the result of applying the operator to corresponding elements.
- As a simple example, A + B is an array each of whose elements is the sum of the corresponding elements of A and B.

Dynamic Arrays

Several languages, allow strings—arrays of characters—to change size after elaboration time string variables in Java and C# are references to immutable string objects:

```
String s = "short"
...
s = s + " but sweet"; // + is the concatenation operator
```

Here the declaration String s introduces a string variable, which we initialize with a reference to the constant string short.

Memory Layout

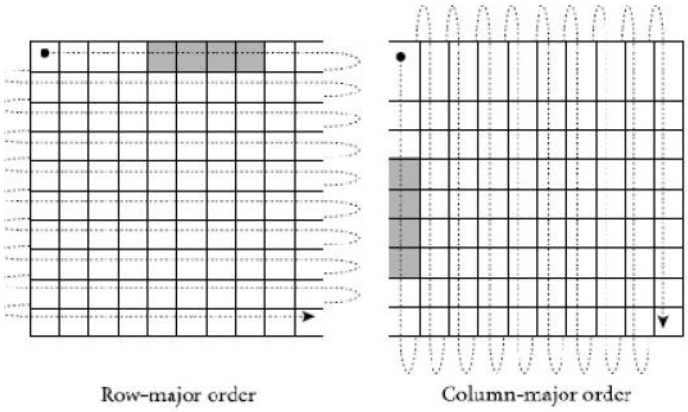
- Arrays in most language implementations are stored in contiguous locations in memory. In a one-dimensional array, the second element of the array is stored immediately after the first (subject to alignment constraints); the third is stored immediately after the second, and so forth.

- For multidimensional arrays, There are two reasonable answers, called *row-major* and *column-major* order. In row- major array layout major order, consecutive locations in memory hold elements that differ by one in the final subscript A[2, 4]

- The advantage of row-major order is that it makes it easy to define a multidimensional array as an array of subarrays, The advantage of row-major order is that it makes it easy to define a multidimensional array as an array of subarrays.

- The difference between row- and column-major layout can be important for programs that use nested loops to access all the elements of a large, multidimensional array.

- Fig: orientation of cache lines for Array layout and cache performance row- and column-major layout of arrays



- If a small array is accessed frequently, all or most of its elements are likely to remain in the cache, and the orientation of cache lines will not matter.

- For a large array, however, many of the accesses that occur during a full-array traversal are likely to result in cache misses, because the corresponding lines have been evicted from the cache since the last traversal.

Dope Vectors

- A dope vector for an array of dynamic shape will contain the lower bound of each dimension and the size of every dimension except the last

- If the language implementation performs dynamic semantic checks for out-of-bounds
- subscripts in array references, then the dope vector will need to contain upper bounds as well.

- Given upper and lower bounds, the size information is redundant, but it is usually included anyway, to avoid computing it repeatedly at run time.

- The dope vector for an array of dynamic shape is generally placed next to the pointer to the array in the fixed-size part of the stack frame. The contents of the dope vector are initialized at elaboration time, or whenever the array changes shape.

5. STRINGS:

In many languages, a string is simply an array of characters. In other languages, strings have special status, with operations that are not available for arrays of other sorts. Particularly powerful string facilities are found in Snobol, Icon, and the various scripting languages. Several languages that do not in general allow arrays to change size dynamically do provide this flexibility for strings. .

First, manipulation of variable-length strings is fundamental to a huge number of computer applications, and in some sense “deserves” special treatment. Second, the fact that strings are

one-dimensional, have onebyte elements, and never contain references to anything else makes dynamic-size strings easier to implement than general dynamic arrays.

Some languages require that the length of a string-valued variable be bound no later than elaboration time, allowing the variable to be implemented as a contiguous array of characters in the current stack frame.

Languages in this category include C, Pascal, and Ada. Pascal and Ada support a few string operations, including assignment and comparison for lexicographic ordering. C, Char* assignment in C hand, provides only the ability to create a pointer to a string literal.

Because of C's unification of arrays and pointers, even assignment is not supported. given the declaration char *s, the statement s = "abc" makes s point to the constant "abc" in static storage. If s is declared as an array, rather than a pointer (char s[4]), then the statement will trigger an error message from the compiler. To assign one array into another in C, the program must copy the elements individually.

Other languages allow the length of a string-valued variable to change over its lifetime, requiring that the variable be implemented as a block or chain of blocks in the heap. All languages a string variable is a *reference* to a string. Assigning a new value to such a variable makes it refer to a different object.
Concatenation and other string operators implicitly create new objects. The space used by objects that are no longer reachable from any variable is reclaimed automatically.

6. SETS:

A programming language set is an unordered collection of an arbitrary number of distinct values of a common type. Sets were introduced by Pascal, and are found in many more recent languages as well. They are a useful form of composite type for many applications. Pascal supports sets of any discrete type, and Set types provides union, intersection, and difference operations:

Ex: var A, B, C : set of char;
D, E : set of weekday;

...

```
A := B + C; (* union; A := {x | x is in B or x is in C} *)
A := B * C; (* intersection; A := {x | x is in B and x is in C} *)
A := B - C; (* difference; A := {x | x is in B and x is not in C} *)
```

There are many ways to implement sets, including arrays, hash tables, and various forms of trees. The most common implementation employs a bit vector whose length (in bits) is the number of distinct values of the base type.

A set of characters, for example (in a language that uses ASCII) would be 128 bits—16 bytes—in length. A one in the *k*th position in the bit vector indicates that the *k*th element of the base type is a member of the set; a zero indicates that it is not.

Operations on bit-vector sets can make use of fast logical instructions on most machines. Union is bit-wise or; intersection is bit-wise and; difference is bit-wise not, followed by bit-wise and.

7. Pointers and recursive types:

A recursive type is one whose objectsmay contain one or more references to other objects of the type. Most recursive types are records, since they need to contain something in addition to the reference, implying the existence of heterogeneous fields. Recursive types are used to build a wide variety of "linked" data structures, including lists and trees.

Syntax and Operations

No aggregate syntax is available for linked data structures in Pascal, Ada, or C;
a tree must be constructed node by node.

To allocate a new node from the heap, Allocating heap nodes the programmer calls a built-in function. In Pascal:

```
new(my_ptr);
```

In Ada:

```
my_ptr := new chr_tree;
```

In C:

```
my_ptr = (struct chr_tree *) malloc(sizeof(struct chr_tree));
```

C's malloc is defined as a library function, not a built-in part of the language (though some compilers recognize and optimize it as a special case); hence the need to specify the size of the allocated object, and to cast the return value to the appropriate type.

C++, Java, and C# replace malloc with a built-in new:

Object-oriented allocation of heap nodes my_ptr = new chr_tree(*arg list*);
In addition to "knowing" the size of the requested type, the C++/Java/C# new will automatically call any user-specified *constructor* (initialization) function, passing the specified argument list. In a similar but less flexible vein, Ada's new may specify an initial value for the allocated object:

```
my_ptr := new chr_tree'(null, null, 'X');
```

Pointers and Arrays in C

Pointers and arrays are closely linked in C. Consider the following declarations.
Array names and pointers
in C

```
int n;
int *a;
int b[10];
```

Now all of the following are valid.

```
1. a = b;  2. n = a[3];
3. n = *(a+3);  4. n = b[3];
5. n = *(b+3);
```

For a one-dimensional array of integers, the corresponding formal parameter may be declared as int a[ ] or int *a. For a two-dimensional array of integers with row-pointer layout, the formal parameter may be declared as int *a[ ] or int **a.

For a two dimensional array with contiguous layout, the formal parameter may be declared as int a[ ][m] or int (*a)[m]. The size of the first dimension is irrelevant; all that is passed is a pointer, and C performs no dynamic checks to ensure that references are within the bounds of the array.

Dangling References

When an object is no longer live, a long-running programneeds to reclaim the object's space. Stack objects are reclaimed automatically as part of the subroutine calling sequence. How are heap objects reclaimed. There are two alternatives.
Languages like Pascal, C, and C++ require the programmer to reclaim an object
Explicit storage reclamation explicitly.
In Pascal:
dispose(my_ptr);
In C:
free(my_ptr);
In C++:
A *dangling reference* is a live pointer that no longer points to a valid object.
In languages like Algol 68 or C, which allow the programmer to create pointers to stack objects, a dangling reference may be created when a subroutine returns while some pointer in a wider scope still refers to a local object of that subroutine.
In a language with explicit reclamation of heap objects, a dangling reference is
created whenever the programmer reclaims an object to which pointers still refer.

Garbage Collection
Explicit reclamation of heap objects is a serious burden on the programmer and a major source of bugs (memory leaks and dangling references). The code required to keep track of object lifetimes makes programs more difficult to design, implement, and maintain. An attractive alternative is to have the language implementation notice when objects are no longer useful and reclaim them automatically.

Tracing Collection
A better definition of a "useful" object is one that can be reached by following a chain of valid pointers starting from something that has a name (i.e., something outside the heap). According to this definition, the blocks in the bottom half of are useless, even though their reference counts are nonzero. Tracing collectors work by recursively exploring the heap, starting from external pointers, to determine what is useful.

Mark-and-Sweep
The classic mechanism to identify useless blocks, under this more accurate definition, is known as *mark-and-sweep*. It proceeds in three main steps, executed by the garbage collector when the amount of free space remaining in the heap falls below some minimum threshold.
1. The collector walks through the heap, tentatively marking every block as "useless."

2. Beginning with all pointers outside the heap, the collector recursively explores all linked data structures in the program, marking each newly discovered block as "useful."
3. The collector again walks through the heap, moving every block that is still marked "useless" to the free list.

Conservative Collection

- The key is to observe that the number of blocks in the heap is much smaller than the number of possible bit patterns in an address. The probability that a word in memory that is not a pointer into the heap will happen to contain a bit pattern that looks like such a pointer is relatively small..

- conservatively, that everything that seems to point to a heap block is in fact a valid pointer, then we can proceed with mark-and-sweep collection. When space runs low, the collector tentatively marks all blocks in the heap as useless.

- It then scans all word-aligned quantities in the stack and in global storage. If any of these "pointers" contains the address of a block in the heap, the collector marks that block as useful. Recursively, the collector then scans all word-aligned quantities in the block and marks as useful any other blocks whose addresses are found therein. Finally the collector reclaims any blocks that are still marked useless.

8.Lists

List is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list. Lists are ideally suited to programming in functional and logic languages, which do most of their work via recursion and higher-order functions

Lists in ML are homogeneous: every element of the list must have the same type. Lisp lists, by contrast, are heterogeneous: any object may be placed in a list, as long as it is never used in an inconsistent fashion.

The different approaches to type in ML and in Lisp lead to Lists in ML and Lisp different implementations. An ML list is usually a chain of blocks, each of which contains an element and a pointer to the next block. A Lisp list is a chain of cons cells, each of which contains *two* pointers, one to the element and one to the next cons cell .

For historical reasons, the two pointers in a cons cell are known as the car and the cdr; they represent the head of the list and the remaining elements, respectively. In both semantics and implementation Clu resembles ML, while Python and Prolog resemble Lisp.

An ML list is en- List notation closed in square brackets, with elements separated by commas: [a, b, c, d]. A Lisp list is enclosed in parentheses, with elements separated by white space: (a b c d).

In both cases, the notation represents a *proper* list: one whose innermost pair consists of the final element and the empty list. In Lisp, it is also possible to construct an *improper* list, whose final pair contains two elements.

In ML the equivalent operations are written as follows. Basic list operations in ML
a :: [b] _->Ë [a, b]
hd [a, b] _->Ë a
hd [ ] _->Ë *run-time exception*
tl [a, b, c] _->Ë [b, c]
tl [a] _->Ë nil
tl [ ] _->Ë *run-time exception*
[a, b] @ [c, d] _-?Ë [a, b, c, d]

9.Files and Input/Output

a. Input/output (I/O) facilities allow a program to communicate with the outside world. Interactive I/O generally implies communication with human users or physical devices, which work in parallel with the running program, and whose input to the program may depend on earlier output from the program

b. Files may be further categorized into those that are *temporary* and those that are *persistent*. Temporary files exist for the duration of a single program run; their purpose is to store information that is too large to fit in the memory available to the program.

c. Persistent files allow a program to read data that existed before the program began running, and to write data that will continue to exist after the program has ended.

d. The principal advantage of language integration is the ability to employ non-subroutine-call syntax and to perform operations that may not otherwise be available to library routines.

e. A purely library-based approach to I/O, on the other hand, may keep a substantial amount of “clutter” out of the language definition.

10.Equality Testing and Assignment

- Data types such as integers, floating-point numbers, or characters, equality testing and assignment are relatively operations, problem of comparing two character strings. Should the expression s = t determine whether s and t
  - are aliases for one another?
  - occupy storage that is bit-wise identical over its full length?
  - contain the same sequence of characters?
  - would appear the same if printed?

- In imperative programming languages assignment operations may also be deep or shallow. Under a reference model of variables, a shallow assignment a := b will make a refer to the object to which b refers.

- A deep assignment will create a copy of the object to which b refers, and make a refer to the copy. Under a value model of variables, a shallow assignment will copy the value of b into a, but if that value is a pointer (or a record containing pointers), then the objects to which the pointer(s) refer will not be copied.

- deep assignments are relatively rare. They are used primarily in distributed computing, and in particular for parameter passing in remote procedure call.

UNIT – 4

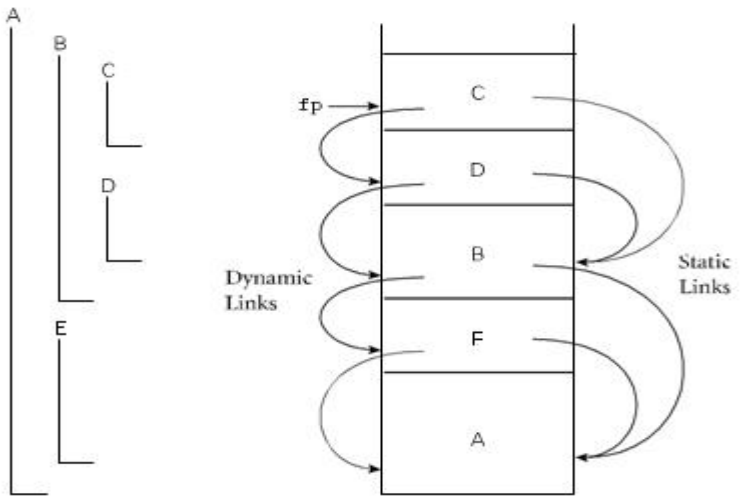
Subroutines and Control Abstraction

- Review of stack layout
- Calling sequences
- Parameter passing
- Generic subroutines and modules
- Exception handling
- Coroutines
- Events.

UNIT 4

Review of Stack Layout

- Layout of run-time stack of run time stack as shown in below figure. Each routine, as it is frame called, is given a new *stack frame* or *activation record*, at the top of the stack.
- This frame may contain arguments and/or return values, bookkeeping information (including the return address and saved registers), local variables, and/or temporaries.
- When a subroutine returns,its frame is popped from the stack.
- At any given time, the *stack pointer* register contains the address of either the last used location at the top of the stack or the first unused location, depending on convention.
- The *frame pointer* register contains an address within the frame.



Static and dynamic links

- In a language with nested subroutines and static scoping (e.g., Pascal, Ada, Static and dynamic links ML, Common Lisp, or Scheme), objects that lie in surrounding subroutines and are thus neither local nor global can be found by maintaining a *static chain* .
- Each stack frame contains a reference to the frame of the lexically surrounding subroutine.
- This reference is called the *static link*. By analogy, the saved value of the frame pointer, which will be restored on subroutine return, is called the *dynamic link*.
- The static and dynamic links may or may not be the same, depending on whether the current routine was called by its lexically surrounding routine, or by some other routine nested in that surrounding routine.

Visibility of nested routines

Whether or not a subroutine is called directly by the lexically surrounding routine, we can be sure that the surrounding routine is active; there is no other way that the current routine could have been visible, allowing it to be called.

EXAMPLE :Consider for example, the subroutine nesting shown in above Figure

If subroutine Visibility of nested routines D is called directly from B, then clearly B's frame will already be on the stack How else could D be called? It is not visible in A or E, because it is nested inside of B. A moment's thought makes clear that it is only when control enters B (placing B's frame on the stack) that D comes into view. It can therefore be called by C, or by any other routine (not shown) that is nested inside of C or D, but only because these are also within B. _

Calling Sequences

- we also mentioned that maintenance of the subroutine call stack is the responsibility of the *calling sequence*—the code executed by the caller immediately before and after a subroutine call.
- *prologue* (code executed at the beginning)
- *epilogue* (code executed at the end) of the subroutine itself.
- Sometimes the term "calling sequence" is used to refer to the combined operations of the caller, the prologue, and the epilogue.

Tasks that must be accomplished on the way into a subroutine include passing parametrs, saving the return address, changing the program counter, changing the stack pointer to allocate space, saving registers (including the frame pointer) that contain important values and that may be overwritten by the callee.

Changing the frame pointer to refer to the new frame, and executing initialization code for any objects in the new frame that require it.

Tasks that must be accomplished on the way out include passing return parameters or function values, executing finalization code for any local objects that require it, deallocating the stack frame.

In general, we will save space if the callee does as much work as possible: tasks performed in the callee appear only once in the target program, but tasks performed in the caller appear at every call site, and the typical subroutine is called in more than one place.

Saving and Restoring Registers

- The ideal approach is to save precisely those registers that are both in use in the caller and needed for other purposes in the callee.

- Because of separate compilation, however, it is difficult to determine this intersecting set. A simpler solution is for the caller to save all registers that are in use, or for the callee to save all registers that it will overwrite.

- Calling sequence conventions for many processors, including the MIPS and x86 registers not reserved for special purposes are divided into two sets of approximately equal size.

- One set is the caller's responsibility, the other is the callee's responsibility. A callee can assume that there is nothing of value in any of the registers in the caller-saves set; a caller can assume that no callee will destroy

The contents of any registers in the callee-saves set. In the interests of code size,

the compiler uses the callee-saves registers for local variables and other long-lived values whenever possible.

It uses the caller-saves set for transient values, which are less likely to be needed across calls. The result of these conventions is that the caller-saves registers are seldom saved by either party: the callee knows that they are the caller's responsibility, and the caller knows that they don't contain anything important.

Maintaining the Static Chain

In languages with nested subroutines, at least part of the work required to maintain the static chain must be performed by the caller, rather than the callee, because this work depends on the lexical nesting depth of the caller. The standard approach is for the caller to compute the callee's static link and to pass it as an extra, hidden parameter.

The following subcases arise.

1. The callee is nested (directly) inside the caller. In this case, the callee's static link should refer to the caller's frame.

2. The callee is $k \geq 0$ scopes "outward"—closer to the outer level of lexical nesting.

In this case, all scopes that surround the callee also surround the caller the caller dereferences its own static link *k* times and passes the result as the callee's static link.

A Typical Calling Sequence

A typical calling sequence The stack pointer (sp) points to the first unused location on the stack The frame pointer (fp) points to a location near the bottom of the frame. Space for all arguments is reserved in the stack, even if the compiler passes some of them in registers .

To maintain this stack layout, the calling sequence might operate as follows.

The caller:

1. saves any caller-saves registers whose values will be needed after the call.

2. Computes the values of arguments and moves them into the stack or registers.

3. Computes the static link and passes it as an extra, hidden argument.

4. Uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the return address on the stack or in a register.

In its prologue, the callee:

1 allocates a frame by subtracting an appropriate constant from the sp.

2. Saves the old frame pointer into the stack, and assigns it an appropriate new value.

After the subroutine has completed, the epilogue:

1. moves the return value (if any) into a register or a reserved location in thestack.

2. Restores callee-saves registers if needed.

3. Restores the fp and the sp.

4. Jumps back to the return address.Finally, the caller moves the return value to wherever it is needed. restores caller-saves registers if needed. _

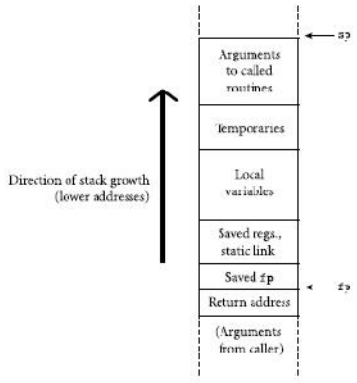


Figure . A typical stack frame.

Special Case Optimizations

Many parts of the calling sequence, prologue, and epilogue can be omitted in common cases. If the hardware passes the return address in a register, then a *leaf routine* can simply leave it there; it does not need to save it in the stack. Likewise it need not save the static link or any caller-saves registers.

A subroutine with no local variables and nothing to save or restore may not even need a stack frame on a RISC machine. The simplest subroutines (e.g., library routines to compute the standard mathematical functions) may not touch memory at all, except to fetch instructions.

Displays

- One disadvantage of static chains is that access to an object in a scope *k* levels out requires that the static chain be dereferenced *k* times.
- If a local object can be loaded into a register with a single memory access, an object *k* levels out will require *k*+1 memory accesses.
- This number can be reduced to a constant by use of a *display*.

Case Studies: C on the MIPS; Pascal on the x86

- Calling sequences differ significantly from machine to machine and even compiler to compiler .
- Some of the most significant differences can be found in a comparison of CISC and RISC conventions.
- Compilers for CISC machines tend to pass arguments on the stack; compilers for RISC machines tend to pass arguments in registers.
- Compilers for CISC machines usually dedicate a register to the frame pointer; compilers for RISC machines often do not.
- Compilers for CISC machines often rely on special purpose instructions to implement parts of the calling sequence; available instructions on a RISC machine are typically much simpler.

The use of the stack to pass arguments reflects the technology of the 1970s, when register sets were significantly smaller and memory access was significantly faster than is the case today. Most CISC instruction sets include push and pop instructions that combine a store or load with automatic update of the stack pointer.

In-Line Expansion

- As an alternative to stack-based calling conventions, many language implementations allow certain subroutines to be expanded in-line at the point of call.
- In-line expansion avoids a variety of overheads, including space allocation, branch delays from the call and return, maintaining the static chain or display, and saving and restoring registers.

- It also allows the compiler to perform code improvements such as global register allocation and common sub expression elimination across the boundaries between subroutines, something that most compilers can't do otherwise.
- In many implementations the compiler chooses which subroutines to expand in-line and which to compile conventionally.

In-lining and recursion

EXAMPLE :General case for recursive subroutines. For the occasional case in which a recursive call is possible but unlikely, it may be desirable to generate a true recursive subroutine, but to expand one instance of it in-line at each call site.

Consider the following C routine for use in hash-table lookup

```
range_t bucket_contents(bucket *b, domain_t x)

{

if (b->key == x)

return b->val;

else if (b->next == 0)

return ERROR;

else

return bucket_contents(b->next, x);

}
```

We can expand this code in-line if we make the nested invocation a true subroutine call. Since most hash chains are only one bucket long, the nested call will usually not occur.

Parameter Passing

- ✓ Most subroutines are parameterized: they take arguments that control certain aspects of their behavior, or specify the data on which they are to operate.
- ✓ Parameter names that appear in the declaration of a subroutine are known as *formal parameters*.
- ✓ Variables and expressions that are passed to a subroutine in a particular call are known as *actual parameters*. We have been referring to actual parameters as *arguments*.
- ✓ In the following two subsections, we discuss the most common parameter-passing *modes*, most of which are implemented by passing values, references, or closures

Most languages use a prefix notation for calls to user-defined subroutines, with the subroutine name followed by a parenthesized argument list.

Parameter Modes:

- ✓ Semantic rules hat govern parameter passing, and that determine the relationship between actual nd formal parameters.
- ✓ Some languages, including C, Fortran, ML, and Lisp, efine a single set of rules, which apply to all parameters.
- ✓ Other languages, including Pascal, Modula, and Ada, provide two or more sets of rules, corresponding to different parameter

Passing a subroutine argument

Suppose for the moment that x is a global variable in a language with a value Passing a subroutine argument model of variables, and that we wish to pass x as a parameter to subroutine p: p(x);

From an implementation point of view, we have two principal alternatives:

1. We may provide p with a copy of x's value
2. We may provide it with x's address.

The two most common parameter-passing modes, called *call by value* and *call by reference*, are designed to reflect these implementations. _

With value parameters, each actual parameter is assigned into the corresponding formal parameter when a subroutine is called; from then on, the two are independent.

With reference parameters, each formal parameter introduces, within the body of the subroutine, a new name for the corresponding actual parameter.

Value and reference parameters

As a simple example, consider the following pseudocode.

Value and reference parameters

```
x : integer — global
procedure foo(y : integer)
y := 3
print x
. . .
x := 2
foo(x)
print x
```

If y is passed to foo by value, then the assignment inside foo has no visible effect—y is private to the subroutine—and the program prints 2 twice. If y is passed to foo by reference, then the assignment inside foo changes x—y is just a local name for x—and the program prints 3 twice. _

Emulating call-by-reference in C

EXAMPLE : To allow a called routine to modify a variable other than an array in the caller's scope, the C programmermust pass the address of the variable explicitly:

```
void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }
swap(&v1, &v2); _
```

Fortran passes all parameters by reference, but it does not require that every actual parameter be an l-value.

If a built-up expression appears in an argument list, the compiler creates a temporary variable to hold the value, and passes this variable by reference.

A Fortran subroutine that needs to modify the values of its formal parameters without modifying its actual parameters must copy the values into local variables and modify those instead.

Call by Sharing

- In languages like Smalltalk, Lisp, ML, and Clu, which use a reference model of variables, an actual parameter is already a reference to an object.
- Instead of passing the value of the actual parameter or a reference to the, actual parameter these languages provide a single parameter-passing mode in which the actual and formal parameters refer to the same object. Clu calls this mode *call by sharing*.

- For variables that are implemented as addresses, call by sharing is usually implemented by passing the address.
- For variables that refer to immutable objects (numbers, characters, etc.) and that are implemented as values, call by sharing is usually implemented by passing the value.
- In Java, parameters of primitive types are passed by value; object parameters are passed by sharing.
- A similar approach is the default in C#, but because the language allows users to create both value and reference types, both cases are considered call by value.

The Ambiguity of Call by Reference

In a language that provides both value and reference parameters, there are two principal reasons why the programmer might choose one over the other.

Two principle ways are:

1. First, if the called routine is supposed to change the value of an actual parameter, then the programmer must pass the parameter by reference
2. Second, the implementation of value parameters requires copying actuals to formals, a potentially time-consuming operation when arguments are large.

Read-Only Parameters

To combine the efficiency of reference parameters and the safety of value parameters, Modula-3 provides a READONLY parameter mode.

Any formal parameter whose declaration is preceded by READONLY cannot be changed by the called routine: the compiler prevents the programmer from using that formal parameter on the left-hand side of any assignment statement, reading it from a file, or passing it by reference to any other subroutine.

Small READONLY parameters are generally implemented by passing a value; larger READONLY parameters are implemented by passing an address. As in Fortran, aModula-3 compiler will create a temporary variable to hold the value of any built-up expression passed as a large READONLY parameter.

Const parameters in C

The equivalent of READONLY parameters is also available in C, which allows any variable or parameter declaration to be preceded by the keyword const.

Example:

Const parameters in C rameters are particularly useful when passing addresses:

```
void append_to_log(const huge_record *r) { ...}
```

append_to_log(&my_record);

Here the keyword const applies to the record to which r points;3 the caller must pass the address of its record explicitly, but can be assured that the callee will not change the record's contents.

One traditional problem with parameter modes—and with the READONLY mode in particular—is that they tend to confuse the key pragmatic issue with two semantic issues: is the callee allowed to change the formal parameter

Parameter Modes in Ada

Ada provides three parameter-passing modes, called in, out, and in out. In_parameters pass information from the caller to the callee; they can be read by the callee but not written.

Out parameters pass information from the callee to the caller. In Ada 83 they can be written by the callee but not read; in Ada 95 they can be both read and written, but they begin their life uninitialized.

In out parameters pass information in both directions; they can be both read and written. Changes to out or in out parameters always change the actual parameter.

For parameters of scalar and access (pointer) types, Ada specifies that all three modes are to be implemented by copying values.

For these parameters, then, in is call by value, out is what some authors call *call by result* and in out is *call by value/result*

Reference and value/result parameters

EXAMPLE :

x : integer — global

procedure foo(y : integer)

y := 3

print x

. . .

x := 2

foo(x)

print x

We already noted that if y is passed by reference the program will print 3 twice.

If y is passed by value/result, it will print 2 and then 3.

One possible way to hide the distinction between reference and value/result would be to outlaw the creation of aliases, as Euclid does.

Ada takes a simpler stack: a program that can tell the difference between value and address-based implementations of (nonscalar, nonpointer) in out parameters is said to be "erroneous"—incorrect, but in a way that the language implementation is not required to catch.

References in C++

Programmers who switch to C after some experience with Pascal, Modula, or

Ada (or with call by sharing in Java or Lisp) are often frustrated by C's lack of reference parameters. As noted above, one can always arrange to modify an object by passing its address, but then the formal parameter is a pointer.

Reference parameters in C++

Consider the function:

```
void swap(int &a, int &b)
{ int t = a; a = b; b = t; }
```

In the code of this swap routine, a and b are ints, not pointers to ints; no dereferencing is required. Moreover, the caller passes as arguments the variables whose values are to be swapped, rather than passing their addresses. _

As in C, a C++ parameter can be declared to be const to ensure that it is not modified. For large types, const reference parameters in C++ provide the same combination of speed and safety found in the READONLY parameter.

References as aliases in C++

EXAMPLE Any variable can be declared to be a reference:

```
int i;
int &j = i;
...
i = 2;
```

```
j = 3;

cout << i; // prints 3
```

Here j is a reference to (an alias for) i. The initializer in the declaration is required; it identifies the object for which j is an alias. Moreover it is not possible later to change the object to which j refers; it will always refer to i.

Closures as Parameters:

A closure (a reference to a subroutine, together with its referencing environment) may be passed as a parameter for any of several reasons.

Subroutines as parameters in Pascal

: In Standard Pascal one might write the following procedure

```
apply_to_A(function f(n : integer) : integer;

var A : array [low..high : integer] of integer);

var i : integer;

begin

for i := low to high do A[i] := f(A[i]);

end;
```

Early versions of Pascal did not include the full header of the subroutine parameter to which it was being passed.

This omission made it difficult or impossible to check at compile time to make sure that the actual and formal parameters expected the same number and types of arguments.

The situation in Fortran is similar: Fortran 77 allows a subroutine to be passed as a parameter but cannot check statically for consistent use.

Subroutine types inModula-2

Several languages provide first-class subroutine *types*, supporting not only

subroutine parameters, but also subroutine variables. In Modula-2 we could

write the following.

EXAMPLE First-class subroutines in ML

```
fun apply_to_L(f, l) =

case l of

nil => nil

| h :: t => f(h) :: apply_to_L(f, t);
```

The type of apply_to_L is (’a -> ’b) * ’a list -> ’b list.

C and C++ get by with simple subroutine pointers because they have no nested subroutines. Similarly, Modula-2 can pass simple addresses because it allows only outermost routines to appear as arguments.

Modula-3 is a bit more general: it allows inner subroutines to be passed as parameters but though it also allows subroutines to be returned from function.

Call by Name

Explicit subroutine parameters are not the only language feature that requires a closure to be passed as a parameter.

In general, a language implementation must pass a closure whenever the eventual use of the parameter requires the restoration of a previous referencing environment.

	implementation mechanism	permissible operations	change to actual?	alias?
value	value	read, write	no	no
`in, const`	value or reference	read only	no	maybe
`out` (Ada)	value or reference	write only	yes	maybe
value/result	value	read, write	yes	no
`var, ref`	reference	read, write	yes	yes
sharing	value or reference	read, write	yes	yes
`in out` (Ada)	value or reference	read, write	yes	maybe
name (Algol 60)	closure (thunk)	read, write	yes	yes

Figure. Parameter passing modes

Special Purpose Parameters

In this subsection we examine other aspects of parameter passing.

Conformant Arrays

- In several languages, the rules for parameters are looser than they are for variables. A formal array parameter whose shape is finalized at run time is called a *conformant*, or *open*, array parameter.
- Because it passes arrays as pointers, C allows actual parameters of different shapes to be passed through the same parameter, but without any runtime checks to ensure that references by the called routine are within the bounds of the actual array.

Default (Optional) Parameters

- We also noted that the same effect can be achieved with *default* parameters.
- A default parameter is one that need not necessarily be provided by the caller; if it is missing, then a preestablished default value will be used instead.

Named Parameters

- In all of our discussions so far we have been assuming that parameters are *positional*: the first actual parameter corresponds to the first formal parameter, the second actual to the second formal, and so on.
- In some languages, including Ada, Common Lisp, Fortran 90, Modula-3, and Python, this need not be the case.
- These languages allow parameters to be *named*. Named parameters (also called *keyword* parameters) are particularly useful in conjunction with default parameters.
- Positional notation allows us to write put(37, 4) to print "37" in a four-column field, but it does not allow us to print in octal in a field of default width.

Variable Numbers of Arguments

Lisp, Python, and C and its descendants are unusual in that they allow the user to define subroutines that take a variable number of arguments.

In C, printf can be declared as follows.

```
int printf(char *format, ...)
{ ...
}
```

The ellipsis (...) in the function header is a part of the language syntax. It indicates

that there are additional parameters following the format, but that their information on parameter types) in C.

Like C and C++, C# and recent versions of Java support variable numbers of parameters, but unlike their parent languages they do so in a type safe manner, by requiring all trailing parameters to share a common type. In Java, for example,

Function Returns

- Many languages place restrictions on the types of objects that can be returned from a function. In Algol 60 and Fortran, a functionmust return a scalar value.

- In Pascal and early versions of Modula-2, it must return a scalar or a pointer. Most imperative languages are more flexible: Algol 68, Ada, C, and many implementations of Pascal allow functions to return values of composite type.
- Modula-3 and Ada 95 allow a function to return a subroutine, implemented as a closure.
- C has no closures, but it allows a function to return a pointer to a subroutine. In functional languages such as Lisp and ML, returning a closure is commonplace.
- The syntax by which a function indicates the value to be returned varies greatly.
- In languages like Lisp, ML, and Algol 68, which do not distinguish between expressions and statements, the value of a function is simply the value of its body, which is itself an expression.
- In several early imperative languages, including Algol 60, Fortran, and Pascal, a function specifies its return value by executing an assignment statement whose left-hand side is the name of the function.
- This approach has an unfortunate interaction with the usual static scope rules: the compiler must forbid any immediately nested declaration

Generic Subroutines and Modules

- Subroutines provide a natural way to perform an operation for a variety of different m object (parameter) values. In large programs, the need also often arises to perform an operation for a variety of different object *types*.
- An operating system, for example, tends to make heavy use of queues, to hold processes, memory descriptors, file buffers, device control blocks, and a host of other objects
- The characteristics of the queue data structure are independent of the characteristics of the items placed in the queue.
- Unfortunately, the standard mechanisms for declaring enqueue and dequeue subroutines in most languages require that the type of the items be declared, statically

Implicit parametric polymorphism

Languages that provide generics include Ada, C++ (which calls them *templates*), Clu, Eiffel, Modula-3, Java, and C#. Generic modules or classes are particularly valuable for creating *containers*:

Generic queues in Ada and C++

Data abstractions that hold a collection of objects, but whose operations are generally oblivious to the type of those objects. Examples of containers include stack, queue, heap, set, and dictionary (mapping) abstractions, implemented as lists, arrays.

Often, as in the case of a sorting routine, the generic code needs to be able to count on certain minimal properties of the type parameters. Appropriate *constraints* may be specified explicitly (as in Ada) or inferred by the compiler (as in C++).

Exception Handling

- Several times in the preceding chapters and sections we have referred to *exception handling* mechanisms.
- We have delayed detailed discussion of these mechanisms until now because exception handling generally requires the language implementation to "unwind" the subroutine call stack.
- An exception can be defined as an unexpected—or at least unusual—condition that arises during program execution, and that cannot easily be handled in the local context.
- It may be detected automatically by the language implementation, or the program may *raise* it explicitly.
- The most common exceptions are various sorts of run-time errors. In an I/O library, for example, an input routine may encounter the end of its file before it can read a requested value, or it may find punctuation marks or letters on the input when it is expecting digits.

To cope with such errors without an exception-handling mechanism, the programmer has basically three options, none of which is entirely satisfactory:

1. "Invent" a value that can be used by the caller when a real value could not be returned.

2. Return an explicit "status" value to the caller, who must inspect it after every call. The status may be written into an extra, explicit parameter, stored in a global variable, or encoded as otherwise invalid bit patterns of a function's regular return value.

3. Pass a closure (in languages that support them) for an error-handling routine that the normal routine can call when it runs into trouble.

The first of these options is fine in certain cases but does not work in the general case. Options 2 and 3 tend to clutter up the program, and impose overhead that we should like to avoid in the common case.

The tests in option 2 are particularly offensive: they obscure the normal flow of events in the common case. Because they are so tedious and repetitive, they are also a common source of errors; one can easily forget a needed test

Exception-handling mechanisms address these issues by moving error-checking code "out of line," allowing the normal case to be specified simply, and arranging for control to branch to a *handler* when appropriate.

Defining Exceptions

In many languages, including Clu, Ada, Modula-3, Python, Java, C#, and ML, most dynamic semantic errors result in exceptions, which the program can then catch.The programmer can also define additional, application-specific exceptions.

Examples of predefined exceptions include arithmetic overflow, division by zero, end-of-file on input, subscript and subrange errors, and null pointer dereference.

The rationale for defining these as exceptions (rather than as fatal errors) is that they may arise in certain valid programs. Some other dynamic errors (e.g., return programmer-defined.

The signal library provided by many C and C++ implementations is independent of language-level exceptions; it allows a program to bind handlers dynamically to certain exceptions detected by the operating system.

EXAMPLE

In Ada, some of the predefined exceptions can be *suppressed* by means of a pragma.In Ada, exception is a built-in type; an exception is simply an object of this declare empty_queue : exception;In Modula-3, exceptions are another “kind” of object, akin to constants, types, variables, or subroutines:

EXCEPTION empty_queue;

In Python, C++, Java, and C#, an exception is an ordinary object, in the object oriented

sense of the word—a value of some class type:

class empty_queue { };

Parameterized exceptions C++/Java/C# and ML

Most languages allow an exception to be “parameterized” so the code that raises the exception can pass information to the code that handles it. In the “parameters” of an exception are naturally expressed as the fields of the class or constructor:

```
class duplicate_in_set { // C++

item dup; // element that was inserted twice

};

...
```

```
throw duplicate_in_set(d);

exception duplicate_in_set of item; (* ML *)

...

raise duplicate_in_set(d);
```

Exception Propagation

In most languages, including Ada, Clu, Modula-3, Python, C++, and Java, an, exception handler is attached to a statement to a list of statements.

In Ada it Exception handler in Ada looks like this: with text_IO; -- import I/O routines (and exceptions)

```
procedure read_rec ... is

begin

...

begin

...

-- potentially complicated sequence of operations

-- involving many calls to text_IO.get

...

exception

when end_error => ...

-- handler to catch any attempt to read past end-of-file

-- in any of the I/O calls

end;

...

end read_rec;
```

Here we have hypothesized a subroutine to read a record from a file. If the file has been corrupted, it may end in the middle of a record. Rather than check for end-of-file at every read (get) operation, we can place the entire series of reads inside a begin...end block that is protected

by a single handler. which is declared in package text_IO. In general, the exception part of a begin...end block can have an arbitrary number of handlers, each for a different exception.

The syntax of the handlers resembles that of an Ada case statement. As in a case statement, the final when clause can be written to catch all

unnamed exceptions:

when others => ... _

Exception handler in C++

EXAMPLE Syntax in other languages is similar. In C++:

try {

...

// protected block of code

...

} catch(end_of_file) {

...

} catch(io_error e) {

// handler for any io_error other than end_of_file

...

} catch(...) {

// handler for any exception not previously named

// (in this case, the triple-dot ellipsis is a valid C++ token;

// it does not indicate missing code)

}

Implementation of Exceptions

- The most obvious implementation for exceptions maintains a linked-list stack. Stacked exception handlers of handlers.

- When control enters a protected block, the handler for that block is added to the head of the list.

- When an exception arises, either implicitly or asa result of a raise statement, the language run-time system pops the innermost handler off the list and calls it.

- The handler begins by checking to see if itmatches the exception that occurred; if not, it simply reraises it:

EX :if exception matches duplicate in set

if exception matches end of file

. . .

elsif exception matches io error

. . .

else

. . . — “catch-all” handler _

Problems :

- The problem with this implementation is that it incurs run-time overhead in the common case.
- Every protected block and every subroutine begins with code to push a handler onto the handler list, and ends with code to pop it back off the list. We can usually do better.
- The only real purpose of the handler list is to determine which handler is active. Since blocks of source code tend to translate into contiguous blocks of machine-language instructions, we can capture the correspondence between handlers and protected blocks in the form of a table generated at compile time.
- Each entry in the table contains two fields: the starting address of a block of code and the address of the corresponding handler.

- The table is sorted on the first field. When an exception occurs, the language run-time system performs binary search in the table, using the program counter as key, to find the handler for the current block.
- If that handler reraises the exception, the process repeats: handlers themselves are blocks of code, and can be found in the table. The only subtlety arises in the case of the implicit handler

Exception Handling without Exceptions

It is worth noting that exceptions can sometimes be simulated in a language that does not provide them as a built-in.

We noted that Pascal permits gotos to labels outside the current subroutine, that Algol 60 allows labels to bepassed as parameters, and that PL/I allows them to be mechanisms permit the program to escape from a deeply nested context, but in a very unstructured way.

UNIT – 5

Data Abstraction and Object Orientation

- Object oriented programming
- Encapsulation and Inheritance
- Initialization and finalization;
- Dynamic method binding
- Multiple inheritances
- Object oriented programming revisited

UNIT 5

Object oriented programming

Object-oriented programming (**OOP**) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are instances of classes, are used to interact with one another to design applications and computer programs.

An object-oriented program may be viewed as a collection of interacting objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to "carry their own operators around with them" (or at least "inherit" them from a similar object or class) - except when they have to be serialized.

Simple, non-OOP programs may be one "long" list of statements (or commands). More complex programs will often group smaller sections of these statements into functions or subroutines each of which might perform a particular task. With designs of this sort, it is common for some of the program's data to be 'global', i.e. accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

In contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by calling specially written functions, commonly called methods, which are bundled in with the data. These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is called an object. The practice of using subroutines to examine or modify certain kinds of data was also used in non-OOP modular programming, well before the widespread use of object-oriented programming.

Encapsulation and Inheritance

In a programming languages, encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

A language mechanism for restricting access to some of the object's components.
A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

Some programming language researchers and academics use the first meaning alone or in combination with the second as a distinguishing feature of object oriented programming, while other programming languages which provide lexical closures view encapsulation as a feature of the language orthogonal to object orientation.

The second definition is motivated by the fact that in many OOP languages hiding of components is not automatic or can be overridden; thus, information hiding is defined as a separate notion by those who prefer the second definition.Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. A benefit of encapsulation is that it can reduce system complexity, and thus increases robustness, by allowing the developer to limit the interdependencies between software components.Almost always, there is a way to override such protection – usually via reflection API (Ruby, Java, C#, etc.), sometimes by mechanism like name mangling (Python), or special keyword usage like friend in C++.

Below is an example in C# that shows how access to a data field can be protected through the use of a private keyword:

```
namespace Encapsulation
{
    class Program
    {
        public class Account
        {
```

```
        private decimal accountBalance = 500.00m;

        public decimal CheckBalance()
        {
            return accountBalance;
        }
    }

    static void Main()
    {
        Account myAccount = new Account();
        decimal myBalance = myAccount.CheckBalance();

        // This Main method can check the balance via the public
        // "CheckBalance" method provided by the "Account" class
        // but it cannot manipulate the value of "accountBalance"
    }
  }
}
```

Initialization and finalization

In object-oriented programming, a constructor (sometimes shortened to ctor) in a class is a special type of subroutine called at the creation of an object. It prepares the new object for use, often accepting parameters which the constructor uses to set any member variables required when the object is first created. It is called a constructor because it constructs the values of data members of the class.

Types of constructors

Parameterized constructors

Constructors that can take arguments are termed as parameterized constructors. The number of arguments can be greater or equal to one(1). For example

```
class example
{
    int p, q;
  public:
    example(int a, int b);                    //parameterized constructor
};
example :: example(int a, int b)
{
    p = a;
    q = b;
}
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.The method of calling the constructor implicitly is also called the shorthand method

```
example e = example(0, 50);                  //explicit call

example e(0, 50);                            //implicit call
```

Default constructors

If the programmer does not supply a constructor for an instantiable class, a typical compiler will provide a default constructor. The behavior of the default constructor is language dependent. It may initialize data members to zero or other same values, or it may do nothing at all.

Copy constructors

Copy constructors define the actions performed by the compiler when copying class objects. A copy constructor has one formal parameter that is the type of the class (the parameter may be a reference to an object).It is used to create a copy of an existing object of the same class. Even though both classes are the same, it counts as a conversion constructor.A constructor is a special type of method.

Dynamic method binding

The property of object-oriented programming languages where the code executed to perform a given operation is determined at run time from the class of the operand(s) (the receiver of the message). There may be several different classes of objects which can receive a given message. An expression may denote an object which may have more than one possible class and that class can only be determined at run time. New classes may be created that can receive a particular message, without changing (or recompiling) the code which sends the message. An class may be created that can receive any set of existing messages.

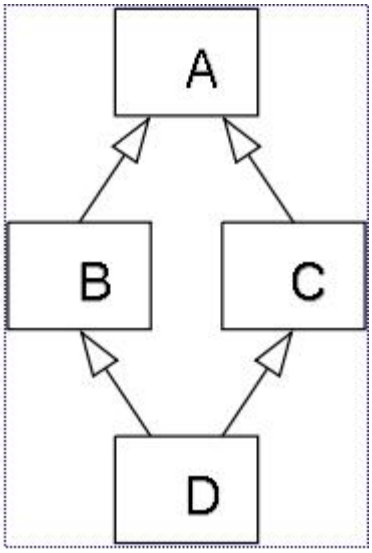
Multiple inheritances

Multiple inheritance is a feature of some object-oriented computer programming languages in which a class can inherit characteristics and features from more than one superclass. It is distinct to single inheritance, where a class may only inherit from one particular superclass.

The "diamond problem" (sometimes referred to as the "deadly diamond of death"[3]) is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C?

For example, in the context of GUI software development, a class Button may inherit from both classes Rectangle (for appearance) and Clickable (for functionality/input handling), and classes Rectangle and Clickable both inherit from the Object class. Now if the equals method is called for a Button object and there is no such method in the Button class but there is an overridden equals method in both Rectangle and Clickable, which method should be eventually called?

It is called the "diamond problem" because of the shape of the class inheritance diagram in this situation. In this article, class A is at the top, both B and C separately beneath it, and D joins the two together at the bottom to form a diamond shape.


A
B
C
D

UNIT – 6

Functional Languages, and Logic Languages

- Functional Languages
  - Origins
  - Concepts
  - A review/overview of scheme
  - Evaluation order revisited
  - Higher-order functions
  - Functional programming in perspective
- Logic Languages
  - Concepts
  - Prolog
  - Logic programming in perspective.

UNIT 6

Historical Origins

1. To understand the differences among programming models, it can be helpful to consider their theoretical roots, all of which predate the development of electronic computers.
2. The imperative and functional models grew out of work undertaken by mathematicians Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, and others in the 1930s.
3. Working largely independently, these individuals developed several very different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics.
4. Over time, these various formalizations were shown to be equally powerful: anything that could be computed in one could be computed in the others.

The goal of early work in computability was not to understand computers. Over time, this work allowed mathematicians to formalize the distinction between a *constructive* proof and a *nonconstructive* proof .

In effect, a program can be seen as a constructive proof of the proposition that, given any appropriate inputs, there exist outputs that are related to the inputs in a particular, desired way.

Euclid'salgorithm, for example, can be thought of as a constructive proof of the proposition that every pair of nonnegative integers has a greatest common divisor.

Logic programming is also intimately tied to the notion of constructive proofs, but at a more abstract level. Rather than write a general constructive proof that works for all appropriate inputs.

Comparing programming models

Logic programming is also intimately tied to the notion of constructive proofs, but at a more abstract level. Rather than write a general constructive proof that works for all appropriate inputs, the logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of

EXAMPLE

To compute the gcd of a and b, check to see if a and b are equal. If so, print one of them and stop. Otherwise, replace the larger one by their difference and repeat. Functional programmer says The gcd of a and b is defined to be a when a and b are equal, and to be the gcdof c and d when a and b are unequal, where c is the smaller of a and b, and ds their difference.

To compute the gcd of a given pair of numbers, expand and simplify this definition until it terminates.the logic programmer says The proposition gcd(a, b, g) is true if (1) a, b, and g are all equal, or (2) there exist numbers c and d such that c is the minimum of a and b (i.e., min(a, b, c)is true), d is their difference (i.e., minus(a, b, d) is true), and gcd(c, d, g)is true.

Functional Programming Concepts

- *functional programming* defines the outputs of program as a mathematical function of t he inputs, with no notion of internal state, and thus no side effects.
- Among the common functional programming languages, Miranda, Haskell, Sisal, pH, and Backus's FP proposal [Bac78] are purely functional; Lisp/Scheme and ML include imperative features.

- To make functional programming practical, functional languages provide a number of features, the following among them, that are often missing in imperative languages.

1. First-class function values and higher-order functions
2. Extensive polymorphism
3. List types and operators
4. Recursion
5. Structured function returns
6. Constructors (aggregates) for structured objects
7. Garbage collection

Lisp was the original functional language and is still the most widely used, several characteristics of Lisp are commonly, though inaccurately, described as though they pertained to functional programming in general.
They include the following.
homogeneity of programs and data: A program in Lisp is itself a list, and can be manipulated with the same mechanisms used to manipulate data.
self-definition: The operational semantics of Lisp can be defined elegantly in terms of an interpreter written in Lisp.
interaction with the user through a "read-eval-print" loop

A Review/Overview of Scheme

Most Scheme implementations employ an interpreter that runs a "read-evalprint"loop. The interpreter repeatedly reads an expression from standard input valuates that expression.

EXAMPLE. If the user types
The read-eval-print loop
(+ 3 4)
the interpreter will print
7
If the user types
7
the interpreter will also print
7
(The number 7 is already fully evaluated.)

To save the programmer the need to type an entire program verbatimat the keyboard ,most Scheme implementations provide a load function that reads (and evaluates) input from a file:

Scheme (like all Lisp dialects) uses *Cambridge Polish* notation for expressions. Parentheses indicate a function application .

EXAMPLE Suppose theSignificance of parentheses user types
((+ 3 4))
When it sees the inner set of parentheses, the interpreter will call the function +,

passing 3 and 4 as arguments.
eval: 7 is not a procedure
Unlike the situation in almost all other programming languages, extra parentheses change the semantics of Lisp/Scheme programs:
(+ 3 4) _⇒ 7
((+ 3 4)) _⇒ *error*
Here the _⇒ means "evaluates to." This symbol is not a part of the syntax of Scheme itself. _

Dynamic typing

Though every expression has a type in Scheme, that type is generally not determined until run time.

.The expression
(if (> a 0) (+ 2 3) (+ 2 "foo"))
will evaluate to 5 if a is positive but will produce a run-time type clash error if a is negative or zero. More significantly,functions that make sense for arguments of multiple types are implicitly polymorphic:
(define min (lambda (a b) (if (< a b) a b)))

Type predicates

EXAMPLE User-defined functions can implement their own type checks using predefined Type predicates *type predicate* functions:
(boolean? x) ; is x a Boolean?
(char? x) ; is x a character?
(string? x) ; is x a string?
(symbol? x) ; is x a symbol?
(number? x) ; is x a number?
(pair? x) ; is x a (not necessarily proper) pair?
(list? x) ; is x a (proper) list?
(This is not an exhaustive list.) _

EXAMPLE In particular, identifiers Liberal syntax for symbols are permitted to contain a wide variety of punctuation marks:
(symbol? 'x$_%:&=*!) _⇒ #t
The symbol #t represents the Boolean value true. False is represented by #f. Note the use here of quote ('); the symbol begins with x. _

Bindings

EXAMPLE: Names can be bound to values by introducing a nested scope.
Nested scopes with let
(let ((a 3)
(b 4)

(square (lambda (x) (* x x)))
(plus +))
(sqrt (plus (square a) (square b)))) _ 5

The special form let takes two arguments. The first of these is a list of pairs. In each pair, the first element is a name and the second is the value that the name is to represent within the second argument to let.

The scope of the bindings produced by let is let's second argument only:

(let ((a 3))
(let ((a 4)
(b a))
(+ a b))) _ 7

Here b takes the value of the *outer* a. The way in which names become visible "all at once" at the end of the declaration list precludes the definition of recursive functions. For these one employs letrec:

(letrec ((fact
(lambda (n)
(if (= n 1) 1
(* n (fact (- n 1)))))))
(fact 5)) _ 120

There is also a let* construct in which names become visible "one at a time" so that later ones can make use of earlier ones, but not vice versa. _

For these Scheme provides a special form called define that has the side effect of creating a global binding for a name:

(define hypot
(lambda (a b)
(sqrt (+ (* a a) (* b b)))))
(hypot 3 4) _

Lists and Numbers

Like all Lisp dialects, Scheme provides a wealth of functions to manipulate lists.

EXAMPLE The three Basic list operations most important are car, which returns the head of a list, cdr ("coulder"), which eturns the rest of the list (everything after the head), and cons, which joins ahead to the rest of a list:

(car '(2 3 4)) _⇒ 2
(cdr '(2 3 4)) _⇒ (3 4)
(cons 2 '(3 4)) _⇒ (2 3 4)
(cdr '(2)) _⇒ ()
(cons 2 3) _⇒ (2 . 3) ; an improper list

Equality Testing and Searching

Scheme provides three different equality-testing un actions. To search for elements in lists, Scheme provides two sets of functions, each of which has variants correspondin the three

different forms of equality. The List search functions functions memq, memv, and member take an element and a list as argument, and return the longest suffix of the list (if any) beginning with the element:

EX:

(memq 'z '(x y z w)) _⇒ (z w)
(memq '(z) '(x y (z) w)) _⇒ #f
(member '(z) '(x y (z) w)) _⇒ ((z) w)

The memq, memv, and member functions perform their comparisons using eq?, eqv?, and equal?, respectively.

They return #f if the desired element is not found. It turns out that Scheme's conditional expressions (e.g., if) treat anything other than #f as true.

Control Flow and Assignment

We have already seen the **EXAMPLE** 10.16 special form if. It has a cousin named cond that re-Multiway conditional expression assembles a more general if. . . elsif. . . else:

(cond
((< 3 2) 1)
((< 4 3) 2)
(else 3)) _⇒ 3

The arguments to cond are pairs. They are considered in order from first to last. The value of the overall expression is the value of the second element of the first pair in which the first element evaluates to #t. If none of the first elements evaluates to #t, then the overall value is #f.

.Many issues related to recursion we do not repeat that discussion here.For programmers who wish to make use of side effects.

Sch **EXAMPLE** signment, sequencing, and iteration constructs. Assignment employs the special Assignment form set! and the functions set-car! and set-cdr!:

(let ((x 2)
(l '(a b)))
(set! x 3)
(set-car! l '(c d))
(set-cdr! l '(e))
... x _⇒ 3
... l _⇒ ((c d) e)

The return values of the various varieties of set! are implementation-dependent.

EXAMPLE 1 Sequencing uses the special form begin:

Sequencing
(begin
(display "hi ")
(display "mom")) _

EXAMPLE Iteration uses the special form do and the function for-each:

Iteration
(define iter-fib (lambda (n)
; print the first n+1 Fibonacci numbers
(do ((i 0 (+ i 1)) ; initially 0, inc'ed in each iteration

```
(a 0 b) ; initially 0, set to b in each iteration
(b 1 (+ a b))) ; initially 1, set to sum of a and b
((= i n) b) ; termination test and final value
(display b) ; body of loop
(display " "))))) ; body of loop
```

Evaluation Order Revisited

- We observed that the subcomponents of many expressions can be evaluated in more than one order. In particular, one can choose to evaluate function arguments before passing them to a function, or to pass them unevaluated.
- The former option is called *applicative-order* evaluation; the latter is called *normal-order* evaluation.
- Like most imperative languages, Scheme uses applicative order in most cases. Normal order, which arises in the macros and call-by name parameters of imperative languages, is available in special cases.

Strictness and Lazy Evaluation

- Evaluation order can have an effect not only on execution speed but on program correctness as well.
- A program that encounters a dynamic semantic error or an infinite regression in an "unneeded" subexpression under applicative-order evaluation may terminate successfully under normal-order evaluation
- A function is said to be *strict* if it requires all of its arguments to be defined, so that its result will not depend on evaluation order.
- A function is said to be *nonstrict* if it does not impose this requirement. A *language* is said to be strict if it requires all functions to be strict.
- A language is said to be nonstrict if it permits the definition of nonstrict functions. Expressions in a strict language can safely be evaluated in applicative order.

Higher-Order Functions

A function is said to be a *higher-order function* (also called a *functional form*) if it takes a function as an argument or returns a function as a result. We have seen several examples already of higher-order functions.

EXAMPLE the Scheme version of map is slightly more general. Like Map function in Scheme for-each, it takes as argument a function and a *sequence* of lists.
. Map calls its function argument on corresponding sets of elements from the lists:

(map * '(2 4 6) '(3 5 7)) _⇒ (6 20 42)

Where for-each is executed for its side effects, and has an implementation dependent return value, map is purely functional: it returns a list composed of the values returned by its function argument.

EXAMPLE Suppose, for example, that we Folding (reduction) in Scheme want to be able to "fold" the elements of a list together, using an associative binary
operator:
(define fold (lambda (f l i)
(if (null? l) i ; i is commonly the identity element for f
(f (car l) (fold f (cdr l) i)))))
Now (fold + '(1 2 3 4 5) 0) gives us the sum of the first five natural numbers,
and (fold * '(1 2 3 4 5) 1) gives us their product. _
tions from existing ones:
(define total (lambda (l) (fold + l 0)))
(total '(1 2 3 4 5)) _⇒ 15
(define total-all (lambda (l)
(map total l)))
(total-all '((1 2 3 4 5)
(2 4 6 8 10)
(3 6 9 12 15))) _⇒ (15 30 45)
(define make-double (lambda (f) (lambda (x) (f x x))))
(define twice (make-double +))
(define square (make-double *)) _

Currying:
EXAMPLE A common operation, named for logician Haskell Curry, is to replace a
multi- Partial application with currying argument function with a function that takes a single argument and returns a function that expects the remaining arguments:

(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4) _⇒ 7
(define plus-3 (curried-plus 3))
(plus-3 4) _⇒ 7
Among other things, currying gives us the ability to pass a "partially applied"
function to a higher-order function:
(map (curried-plus 3) '(1 2 3)) _⇒ (4 5 6) _
EXAMPLE It turns out that we can write a general purpose function that "curries" its
General purpose curry function (binary) function argument:
(define curry (lambda (f) (lambda (a) (lambda (b) (f a b)))))
(((curry +) 3) 4) _⇒ 7
(define curried-plus (curry +)) _

Functional Programming in Perspective

Programmers and compilers of a purely functional language can employ *equational reasoning*, in which the equivalence of two expressions at any point in time implies their equivalence at all times.
Unfortunately, there are common Other commonly cited examples of "naturally imperative" idioms include the following.

- *initialization of complex structures:* The heavy reliance on lists in Lisp, ML, and

Haskell reflects the ease with which functions can build new lists out of the components of old lists.

- *summarization:* Many programs include code that scans a large data structure or a large amount of input data, counting the occurrences of various items or patterns
- *in-place mutation:* In programs with very large data sets, one must economize as much as possible on memory usage, to maximize the amount of data that will fit in memory or the cache.

Logic Programming Concepts

- Logic programming systems allow the programmer to state a collection of *axioms* from which theorems can be proven.
- The user of a logic program states a theorem, or *goal*, and the language implementation attempts to find a collection of axioms and inference steps that together imply the goal. Of the several existing logic languages, Prolog is by far the most widely used.

In almost all logic languages, axioms are written in a standard form known Horn clauses as a *Horn clause*. A Horn clause consists of a *head*,2 or *consequent* term *H*, and a *body* consisting of terms *Bi*:

$H \leftarrow B1, B2, \ldots, Bn$

The semantics of this statement are that when the *Bi* are all true, we can deduce that *H* is true as well. When reading aloud, we say "*H*, if *B*1, *B*2, . . . , and *Bn*."
Horn clauses can be used to capture most, but not all, logical statements.

In order to derive new statements, a logic programming system combines sting statements, canceling like terms, through a process known as *resolution*.

If Resolution we know that *A* and *B* imply *C*, for example, and that *C* implies *D*, we can deduce that *A* and *B* imply *D*:
$C \leftarrow A, B$
$D \leftarrow C$
$D \leftarrow A, B$
In general, terms like *A*, *B*, *C*, and *D* may consist not only of constants

Prolog:

Much as a Scheme interpreter evaluates functions in the context of a referencing environment in which other functions and constants have been defined, a Prolog interpreter runs in the context of a *database* of *clauses* (Horn clauses) that are assumed to be true.

Each clause is composed of *terms*, which may be constants, variables, or *structures*. A constant is either an atom or a number. A structure can be thought of as either a logical predicate or a data structure.

EXAMPLE Structures consist of an atom called the *functor* and a list of arguments:
Structures and predicates
rainy(rochester)
teaches(scott, cs254)

bin_tree(foo, bin_tree(bar, glarch))

Prolog requires the opening parenthesis to come immediately after the functor, with no intervening space. Arguments can be arbitrary terms: constants, variables, or structures. Internally, a Prolog implementation can represent a structure using Lisp-like cons cells.

We use the term *predicate* to refer to the combination of a functor and an "arity" (number of arguments). The predicate rainy has arity 1. The predicate teaches has arity 2. _

Resolution and Unification

The *resolution principle*, due to Robinson [Rob65], says that if *C*1 and *C*2 areHorn clauses and the head of *C*1 matches one of the terms in the body of *C*2, then we can replace the term in *C*2 with the body of *C*1.

Consider the following exam-Resolution in Prolog ple.

EXAMPLE:

takes(jane_doe, his201).
takes(jane_doe, cs254).
takes(ajit_chandra, art302).
takes(ajit_chandra, cs254).
classmates(X, Y) :- takes(X, Z), takes(Y, Z).

Here if we let X be jane_doe and Z be cs254, we can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause, yielding the new rule classmates(jane_doe, Y) :- takes(Y, cs254).

In other words, Y is a classmate of jane_doe if Y takes cs254.

The pattern-matching process used to associate X with jane_doe and Z with cs254 is known as *unification*. Variables that are given values as a result of unification are said to be *instantiated*. The unification rules for Prolog are as follows.

- _ A constant unifies only with itself
- _ Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively.
- _ A variable unifies with anything. If the other thing has a value, then the variable is instantiated.

Lists:

Like equality checking, list manipulation is a sufficiently common operation in Prolog to warrant its own notation.

The construct [a, b, c] is syntactic sugar List notation in Prolog for the structure .(a, .(b, .(c, []))), where [] is the empty list and . is a built-in cons-like predicate.

This notation should be familiar to users of ML. Prolog adds an extra convenience, however: an optional vertical bar that delimits the "tail" of the list. Using this notation, [a, b, c] could be expressed as

[a | [b, c]], [a, b | [c]], or [a, b, c | []]. The ver
member(X, [X|T]).
member(X, [H|T]) :- member(X, T).
sorted([]). % empty list is sorted

```
sorted([X]). % singleton is sorted
sorted([A, B | T]) :- A =< B, sorted([B | T]).
% compound list is sorted if first two elements are in order and
% remainder of list (after first element) is sorted
```

Here =< is a built-in predicate that operates on numbers. Note that [a, b | c] is the *improper* list .(a, .(b, c)). The sequence of tokens [a | b, c] is syntactically invalid.

Arithmetic:

The usual arithmetic operators are available in Prolog, but they play the role of predicates, not of functions. Thus +(2, 3), which may also be written 2 + 3, Arithmetic and the is predicate is a two-argument structure, not a function call. In particular, it will not unifywith 5:

```
?- (2 + 3) = 5.
No
```

To handle arithmetic, Prolog provides a built-in predicate, is, that unifies its first argument with the arithmetic value of its second argument

```
X = 3
?- X is 1+2.
X = 3 % infix is also ok
?- 1+2 is 4-1.
no % first argument (1+2) is already instantiated
?- X is Y.
<error> % second argument (Y) must already be instantiated
?- Y is 1+2, X is Y.
X = 3
Y = 3 % Y is instantiated by the time it is needed
```

Search/Execution Order:

- So how does Prolog go about answering a query ?What it needs is a sequence of resolution steps that will build the goal out of clauses in the database, or a proof that no such sequence exists. In the realm of formal logic, one can imagine two principal search strategies.
- Start with existing clauses and work forward, attempting to derive the goal. This strategy is known as *forward chaining*.
- Start with the goal and work backward, attempting to "unresolve" it into a set of preexisting clauses. This strategy is known as *backward chaining*.

If the number of existing rules is very large, but the number of facts is small, it is possible for forward chaining to discover a solution more quickly than backward chaining. In most circumstances, however, backward chaining turns out to be more efficient. Prolog is defined to use backward chaining.

Let us number the squares from 1 to 9 in row-major order. Further, let us use the Prolog fact x(n) to indicate that player X has placed a marker in square *n*, and o(m) to indicate that player O has placed a marker in square*m*. For simplicity, let us assume that the computer is player X, and that it is X's turn to move.

We should like to be able to issue a query ?- move(A) that will cause the Prolog interpreter to choose a good square A for the computer to occupy next. Clearly we need to be able to tell whether three given squares lie in a row. One way to express this is

```
ordered_line(1, 2, 3). ordered_line(4, 5, 6).
ordered_line(7, 8, 9). ordered_line(1, 4, 7).
ordered_line(2, 5, 8). ordered_line(3, 6, 9).
ordered_line(1, 5, 9). ordered_line(3, 5, 7).
line(A, B, C) :- ordered_line(A, B, C).
line(A, B, C) :- ordered_line(A, C, B).
line(A, B, C) :- ordered_line(B, A, C).
line(A, B, C) :- ordered_line(B, C, A).
line(A, B, C) :- ordered_line(C, A, B).
line(A, B, C) :- ordered_line(C, B, A).
```

It is easy to prove that there is no winning strategy for tic-tac-toe: either player can force a draw. Let us assume, however, that our program is playing against a less-than-perfect opponent.

Logic Languages

Imperative Control Flow

- We have seen that the ordering of clauses and of terms in Prolog is significant, with ramifications for efficiency, termination, and choice among alternatives.
- In addition to simple ordering, Prolog provides the programmer with several explicit control-flow features. The most important of these features is known as the *cut*.
- The cut is a zero-argument predicate written as an exclamation point: !. As a subgoal it always succeeds, but with a crucial side effect.
- It commits the interpreter to whatever choices have beenmade since unifying reconsideration of the prime(a) subgoal, even though that subgoal is doomed to fail.

We can save substantial time by cutting off all further searches for a after the first is found:

```
member(X, [X|T]) :- !.
member(X, [H|T]) :- member(X, T).
```

The cut on the right-hand side of the first rule says that if X is the head of L, we should not attempt to unify member(X, L) with the left-hand side of the second rule; the cut commits us to the first rule. _

EXAMPLE An alternative way to ensure that member(X, L) succeeds no more than once Not and its implementation is to embed a use of not in the second clause:

```
member(X, [X|T]).
member(X, [H|T]) :- not(X = H), member(X, T).
```

This code will display the same high-level behavior but is slightly less efficient: now the interpreter will actually consider the second rule, abandoning it only after (re)unifying X with H and reversing the sense of the test.

It turns out that not is actually implemented by a combination of the cut and two other built-in predicates, call and fail:

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

In principle, it is possible to replace all uses of the cut with uses of not—to confine the cut to the implementation of not. Doing so often makes a program easier to read.

Logic Programming in Perspective

In the abstract, logic programming is a very compelling idea: it suggests a model of computing in which we simply list the logical properties of an unknown value.

Parts of Logic Not Covered

Horn clauses do not capture all of first-order predicate calculus. In particular, they cannot be used to express statements whose clausal form includes a disjunction with more than one nonnegated term.

Execution Order

While logic is inherently declarative, most logic languages explore the tree of possible resolutions in deterministic order. It also provides predicates, including assert, retract, and call,to manipulate its database explicitly during execution.

Negation and the "ClosedWorld" Assumption

A collection of Horn clauses, such as the facts and rules of a Prolog database, constitutes a list of things assumed to be true. It does not include any things assumed to be false.

What Is a Scripting Language?

Modern scripting languages have two principal sets of ancestors. In one set are the command interpreters or "shells" of traditional batch and "terminal" (command-line) computing.

In the other set are various tools for text processing and report generation. Examples in the first set include IBM's JCL, the MSDOS command interpreter, and the Unix sh and csh shell families.

Common Characteristics:

While it is difficult to define scripting languages precisely, there are several characteristics that they tend to have in common.

1. *Both batch and interactive use*. A few scripting languages (notably Perl) use a just-in-time compiler that insists on reading the entire source program before it produces any output.
2. *Economy of expression*. To support both rapid development and interactive use, scripting languages tend to require a minimum of "boilerplate."
3. *Lack of declarations; simple scoping rules*. Most scripting languages dispense with declarations, and provide simple rules to govern the scope of names.
4. *Flexible dynamic typing*. In keeping with the lack of declarations, most scripting languages are dynamically typed
5. *Easy access to other programs*. Most programming languages provide a way to ask the underlying operating system to run another program, or to perform some operation directly.
6. *Sophisticated pattern matching and string manipulation*. In keeping with their text processing and report generation ancestry, and to facilitate the manipulation of textual input and output for external programs.
7. *High-level data types*. High-level data types like sets, bags, dictionaries, lists, and tuples are increasingly common in the standard library packages of conventional programming language.

UNIT – 7

Concurrency

- Background and motivation
- Concurrency programming fundamentals
- Implementing synchronization
- Language-level mechanisms
- Message passing

UNIT 7

Background and motivation

Erlang's main strength is support for concurrency. It has a small but powerful set of primitives to create processes and communicate among them. Processes are the primary means to structure an Erlang application. Erlang processes loosely follow the communicating sequential processes (CSP) model. They are neither operating system processes nor operating system threads, but lightweight processes. Like operating system processes (but unlike operating system threads) they have no shared state between them. The estimated minimal overhead for each is 300 words;[11] thus many of them can be created without degrading performance: a benchmark with 20 million processes has been successfully performed. Erlang has supported symmetric multiprocessing since release R11B of May 2006.

Inter-process communication works via a shared-nothing asynchronous message passing system: every process has a “mailbox”, a queue of messages that have been sent by other processes and not yet consumed. A process uses the `receive` primitive to retrieve messages that match desired patterns. A message-handling routine tests messages in turn against each pattern, until one of them matches. When the message is consumed and removed from the mailbox the process resumes execution. A message may comprise any Erlang structure, including primitives (integers, floats, characters, atoms), tuples, lists, and functions.

Concurrency programming fundamental

- SCHEDULERS give us the ability to "put a thread/process to sleep" and run something else on its process/processor
    - start with coroutines
    - make uniprocessor run-until-block threads
    - add preemption
    - add multiple processors

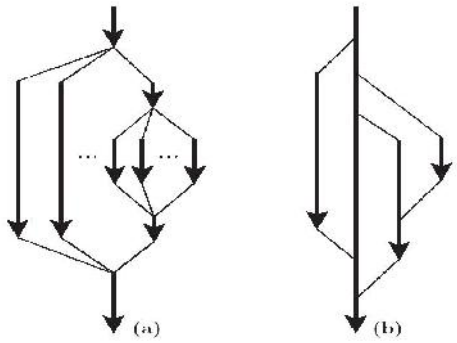


Figure 12.6: **Lifetime of concurrent threads.** With `co-begin`, parallel loops, or launch-at-elaboration (a), threads are always properly nested. With `fork/join` (b), more general patterns are possible.

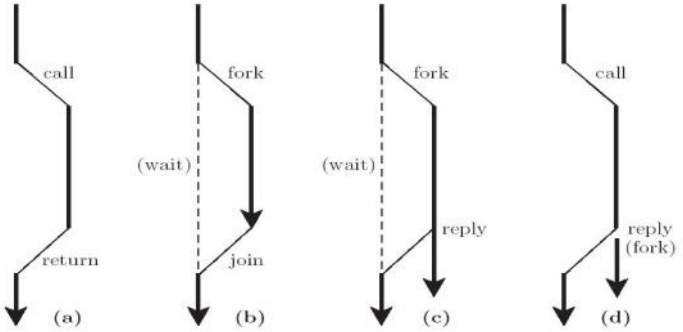


Figure 12.7: **Threads, subroutine calls, and early reply.** Conventionally, subroutine calls are conceptualized as using a single thread (a). Equivalent functionality can be achieved with separate threads (b). Early reply (c) allows a forked thread to continue execution after "returning" to the caller. To avoid creation of a callee thread in the common case, we can wait until the reply to do the fork (d).

- Coroutines
  - Multiple execution contexts, only one of which is active
  - Transfer (other):
    - save all callee-saves registers on stack, including ra and fp
    - *current := sp
    - current := other
    - sp := *current
    - pop all callee-saves registers (including ra, but NOT sp!)
    - return (into different coroutine!)
  - Other and current are pointers to CONTEXT BLOCKs
    - Contains sp; may contain other stuff as well (priority, I/O status, etc.)

- No need to change PC; always changes at the same place
- Create new coroutine in a state that looks like it's blocked in transfer. (Or maybe let it execute and then "detach". That's basically early reply)

- Run-until block threads on a single process
  - Need to get rid of explicit argument to transfer
  - Ready list data structure: threads that are runnable but not running procedure reschedule:

    t : cb := dequeue(ready_list) transfer(t)
  - To do this safely, we need to save 'current' somewhere - two ways to do this:
    - Suppose we're just relinquishing the processor for the sake of fairness (as in MacOS or Windows 3.1):

      procedure yield:enqueue (ready_list, current)reschedule
    - Now suppose we're implementing synchronization:

      sleep_on(q)

      enqueue(q, current)

      reschedule
  - Some other thread/process will move us to the ready list when we can continue

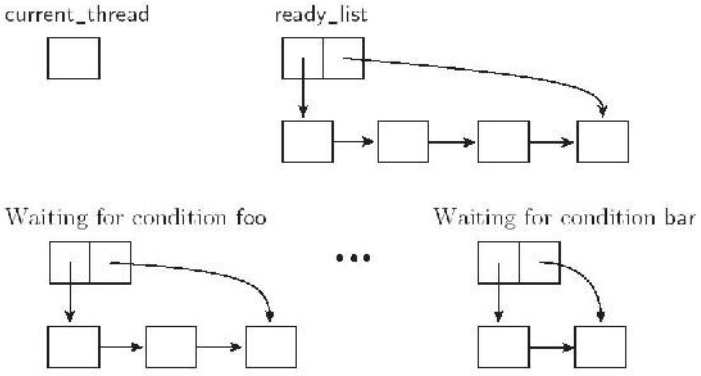


Figure 12.9: **Data structures of a simple scheduler.** A designated `current_thread` is running. Threads on the ready list are runnable. Other threads are blocked, waiting for various conditions to become true. If threads run on top of more than one OS-level process, each such process will have its own `current_thread` variable. If a thread makes a call into the operating system, its process may block in the kernel.

- Preemption
  - Use timer interrupts (in OS) or signals (in library package) to trigger involuntary yields
  - Requires that we protect the scheduler data structures:

```
procedure yield:
disable_signals
enqueue(ready_list, current)
Reschedule
re-enable_signals
```

  - Note that reschedule takes us to a different thread, possibly in code other than yield Invariant: EVERY CALL to reschedule must be made with signals disabled, and must re-enable them upon its return

```
disable_signals
if not <desired condition>
sleep_on <condition queue>
re-enable signals
```

- Multiprocessors
  - Disabling signals doesn't suffice:

```
procedure yield:
 disable_signals
acquire(scheduler_lock)       // spin lock
enqueue(ready_list, current)
reschedule
release(scheduler_lock)
re-enable_signals
disable_signals
acquire(scheduler_lock)       // spin lock
if not <desired condition>
sleep_on <condition queue>
release(scheduler_lock)
re-enable signals
```

Implementing synchronization

How do we implement synchronization operations like locks? Can build synchronization operations out of atomic reads and writes. There is a lot of literature on how to do this, one algorithm is called the bakery algorithm. But, this is slow and cumbersome to use. So, most machines have hardware support for synchronization - they provide synchronization instructions.

On a uniprocessor, the only thing that will make multiple instruction sequences not atomic is interrupts. So, if want to do a critical section, turn off interrupts before the critical section and turn on interrupts after the critical section. Guaranteed atomicity. It is also fairly efficient. Early versions of Unix did this.

Why not just use turning off interrupts? Two main disadvantages: can't use in a multiprocessor, and can't use directly from user program for synchronization.

Test-And-Set. The test and set instruction atomically checks if a memory location is zero, and if so, sets the memory location to 1. If the memory location is 1, it does nothing. It returns the old value of the memory location. You can use test and set to implement locks as follows:

The lock state is implemented by a memory location. The location is 0 if the lock is unlocked and 1 if the lock is locked.

The lock operation is implemented as while (test-and-set(l) == 1)

The unlock operation is implemented as: *l = 0;

The problem with this implementation is busy-waiting. What if one thread already has the lock, and another thread wants to acquire the lock? The acquiring thread will spin until the thread that already has the lock unlocks it.

What if the threads are running on a uniprocessor? How long will the acquiring thread spin? Until it expires its quantum and thread that will unlock the lock runs. So on a uniprocessor, if can't get the thread the first time, should just suspend. So, lock acquisition looks like this:

```
while (test-and-set(l) == 1) {
  currentThread->Yield();
}
```

Can make it even better by having a queue lock that queues up the waiting threads and gives the lock to the first thread in the queue. So, threads never try to acquire lock more than once.

On a multiprocessor, it is less clear. Process that will unlock the lock may be running on another processor. Maybe should spin just a little while, in hopes that other process will release lock. To evaluate spinning and suspending strategies, need to come up with a cost for each suspension algorithm. The cost is the amount of CPU time the algorithm uses to acquire a lock.

There are three components of the cost: spinning, suspending and resuming. What is the cost of spinning? Waste the CPU for the spin time. What is cost of suspending and resuming? Amount of CPU time it takes to suspend the thread and restart it when the thread acquires the lock.

Each lock acquisition algorithm spins for a while, then suspends if it didn't get the lock. The optimal algorithm is as follows:

If the lock will be free in less than the suspend and resume time, spin until acquire the lock.

If the lock will be free in more than the suspend and resume time, suspend immediately.

Obviously, cannot implement this algorithm - it requires knowledge of the future, which we do not in general have.

How do we evaluate practical algorithms - algorithms that spin for a while, then suspend. Well, we compare them with the optimal algorithm in the worst case for the practical algorithm. What is the worst case for any practical algorithm relative to the optimal algorithm? When the lock become free just after the practical algorithm stops spinning.

What is worst-case cost of algorithm that spins for the suspend and resume time, then suspends? (Will call this the SR algorithm). Two times the suspend and resume time. The worst case is when the lock is unlocked just after the thread starts the suspend. The optimal algorithm just spins until the lock is unlocked, taking the suspend and resume time to acquire the lock. The SR

algorithm costs twice the suspend and resume time -it first spins for the suspend and resume time, then suspends, then gets the lock, then resumes.

What about other algorithms that spin for a different fixed amount of time then block? Are all worse than the SR algorithm.

If spin for less than suspend and resume time then suspend (call this the LT-SR algorithm), worst case is when lock becomes free just after start the suspend. In this case the the algorithm will cost spinning time plus suspend and resume time. The SR algorithm will just cost the spinning time.

If spin for greater than suspend and resume time then suspend (call this the GR-SR algorithm), worst case is again when lock becomes free just after start the suspend. In this case the SR algorithm will also suspend and resume, but it will spin for less time than the GT-SR algorithm

Of course, in practice locks may not exhibit worst case behavior, so best algorithm depends on locking and unlocking patterns actually observed.

Here is the SR algorithm. Again, can be improved with use of queueing locks.

```
notDone = test-and-set(l);
if (!notDone) return;
start = readClock();
while (notDone) {
  stop = readClock();
  if (stop - start >= suspendAndResumeTime) {
    currentThread->Yield();
    start = readClock();
  }
  notDone = test-and-set(l);
```

```
}
```

There is an orthogonal issue. test-and-set instruction typically consumes bus resources every time. But a load instruction caches the data. Subsequent loads come out of cache and never hit the bus. So, can do something like this for inital algorithm:

```
while (1) {
  if !test-and-set(l) break;
  while (*l == 1);
}
```

Are other instructions that can be used to implement spin locks - swap instruction, for example.

On modern RISC machines, test-and-set and swap may cause implementation headaches. Would rather do something that fits into load/store nature of architecture. So, have a non-blocking abstraction: Load Linked(LL)/Store Conditional(SC).

Semantics of LL: Load memory location into register and mark it as loaded by this processor. A memory location can be marked as loaded by more than one processor.

Semantics of SC: if the memory location is marked as loaded by this processor, store the new value and remove all marks from the memory location. Otherwise, don't perform the store. Return whether or not the store succeeded.

Here is how to use LL/SC to implement the lock operation:

```
  while (1) {
    LL  r1, lock
    if (r1 == 0) {
      LI r2, 1
      if (SC r2, lock) break;
```

```
  }
}
```

Unlock operation is the same as before.Can also use LL/SC to implement some operations (like increment) directly. People have built up a whole bunch of theory dealing with the difference in power between stuff like LL/SC and test-and-set.

```
while (1) {
  LL   r1, lock
  ADDI r1, 1, r1
  if (SC r2, lock) break;
}
```

Note that the increment operation is non-blocking. If two threads start to perform the increment at the same time, neither will block - both will complete the add and only one will successfully perform the SC. The other will retry. So, it eliminates problems with locking like: one thread acquires locks and dies, or one thread acquires locks and is suspended for a long time, preventing other threads that need to acquire the lock from proceeding.

Message passing

- Message passing system

Distributed object and remote method invocation systems like ONC RPC, CORBA, Java RMI, DCOM, SOAP, .NET Remoting, CTOS, QNX Neutrino RTOS, OpenBinder, D-Bus, Unison RTOS and similar are message passing systems.

Message passing systems have been called "shared nothing" systems because the message passing abstraction hides underlying state changes that may be used in the implementation of sending messages.

Message passing model based programming languages typically define messaging as the (usually asynchronous) sending (usually by copy) of a data item to a communication endpoint (Actor,

process, thread, socket, etc.). Such messaging is used in Web Services by SOAP. This concept is the higher-level version of a datagram except that messages can be larger than a packet and can optionally be made reliable, durable, secure, and/or transacted.

Messages are also commonly used in the same sense as a means of interprocess communication; the other common technique being streams or pipes, in which data are sent as a sequence of elementary data items instead (the higher-level version of a virtual circuit).

- Synchronous versus asynchronous message passing

Synchronous message passing systems require the sender and receiver to wait for each other to transfer the message. That is, the sender will not continue until the receiver has received the message.

Synchronous communication has two advantages. The first advantage is that reasoning about the program can be simplified in that there is a synchronisation point between sender and receiver on message transfer. The second advantage is that no buffering is required. The message can always be stored on the receiving side, because the sender will not continue until the receiver is ready.

Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready. The advantage of asynchronous communication is that the sender and receiver can overlap their computation because they do not wait for each other.

Synchronous communication can be built on top of asynchronous communication by using a so-called Synchronizer. For example, the α-Synchronizer works by ensuring that the sender always waits for an acknowledgement message from the receiver. The sender only sends the next message after the acknowledgement has been received.

The buffer required in asynchronous communication can cause problems when it is full. A decision has to be made whether to block the sender or whether to discard future messages. If the sender is blocked, it may lead to an unexpected deadlock. If messages are dropped, then communication is no longer reliable.

- Message passing versus calling

Message passing should be contrasted with the alternative communication method for passing information between programs – the Call. In a traditional Call, arguments are passed to the "callee" (the receiver) typically by one or more general purpose registers or in a parameter list containing the addresses of each of the arguments. This form of communication differs from message passing in at least three crucial areas

  - total memory usage
  - transfer time
  - locality

In message passing, each of the arguments has to have sufficient available extra memory for copying the existing argument into a portion of the new message. This applies irrespective of the size of the original arguments – so if one of the arguments is (say) an HTML string of 31,000 octets describing a web page (similar to the size of this article), it has to be copied in its entirety (and perhaps even transmitted) to the receiving program (if not a local program).

By contrast, for the call method, only an address of say 4 or 8 bytes needs to be passed for each argument and may even be passed in a general purpose register requiring zero additional storage and zero "transfer time". This of course is not possible for distributed systems since an (absolute) address – in the callers address space – is normally meaningless to the remote program (however, a relative address might in fact be usable if the callee had an exact copy of, at least some of, the callers memory in advance). Web browsers and web servers are examples of processes that communicate by message passing. A URL is an example of a way of referencing resources that does depend on exposing the internals of a process.

A subroutine call or method invocation will not exit until the invoked computation has terminated. Asynchronous message passing, by contrast, can result in a response arriving a significant time after the request message was sent.

A message handler will, in general, process messages from more than one sender. This means its state can change for reasons unrelated to the behaviour of a single sender or client process. This is in contrast to the typical behaviour of an object upon which methods are being invoked: the

latter is expected to remain in the same state between method invocations. (in other words, the message handler behaves analogously to a volatile object).

- Message passing and locks

Message passing can be used as a way of controlling access to resources in a concurrent or asynchronous system. One of the main alternatives is mutual exclusion or locking. Examples of resources include shared memory, a disk file or region thereof, a database table or set of rows.

In locking, a resource is essentially shared, and processes wishing to access it (or a sector of it) must first obtain a lock. Once the lock is acquired, other processes are blocked out, ensuring that corruption from simultaneous writes does not occur. After the process with the lock is finished with the resource, the lock is then released.

With the message-passing solution, it is assumed that the resource is not exposed, and all changes to it are made by an associated process, so that the resource is encapsulated. Processes wishing to access the resource send a request message to the handler. If the resource (or subsection) is available, the handler makes the requested change as an atomic event, that is conflicting requests are not acted on until the first request has been completed. If the resource is not available, the request is generally queued. The sending programme may or may not wait until the request has been completed.

UNIT – 8

Run-Time Program Management

- Virtual machines
- Late binding of machine code
- Inspection/introspection.

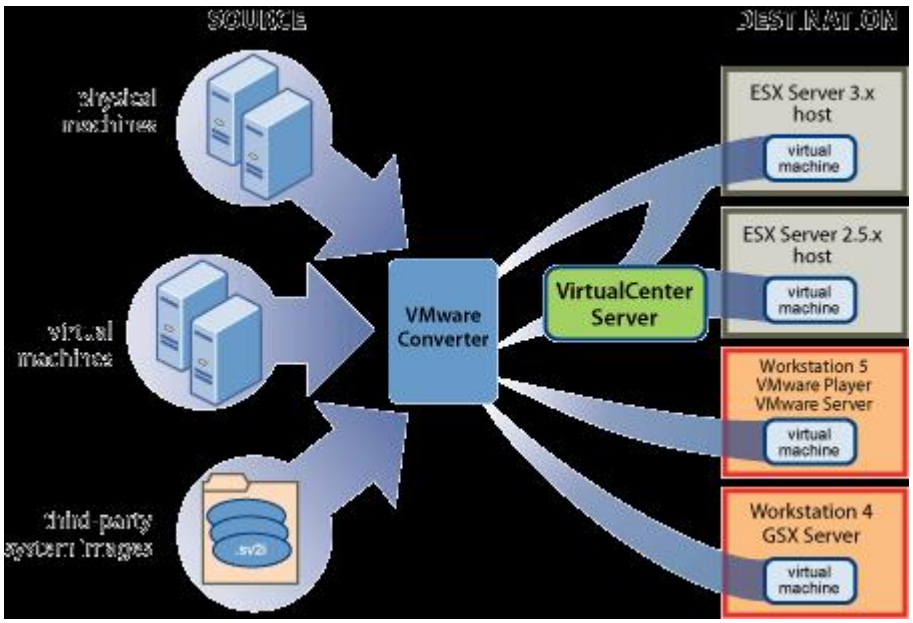
Virtual machines

A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine. Virtual machines are separated into two major categories, based on their use and degree of correspondence to any real machine:

1) A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS) . These usually emulate an existing architecture, and are built with either the purpose of providing a platform to run programs where the real hardware is not available for use (for example, executing software on otherwise obsolete platforms), or of having multiple instances of virtual machines lead to more efficient use of computing resources, both in terms of energy consumption and cost effectiveness (known as hardware virtualization, the key to a cloud computing environment), or both.

2) A process virtual machine (also, language virtual machine) is designed to run a single program, which means that it supports a single process. Such virtual machines are usually closely suited to one or more programming languages and built with the purpose of providing program portability and flexibility (amongst other things). An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine—it cannot break out of its virtual environment.

A virtual machine was originally defined by Popek and Goldberg as "an efficient, isolated duplicate of a real machine". Current use includes virtual machines which have no direct correspondence to any real hardware.



Late binding of machine code

Late binding, A.K.A Dynamic binding is a computer programming mechanism in which the method being called upon an object is looked up by name at runtime. This is informally known as duck typing or name binding.

With Early binding, A.K.A. Static binding, the compilation phase fixes all types of variables and expressions. This is usually stored in the compiled program as an offset in a virtual method table ("v-table") and is very efficient. With late binding the compiler does not have enough information to verify the method even exists, let alone bind to its particular slot on the v-table. Instead the method is looked up by name at runtime.

Inspection/introspection.

Introspection is the ability of a program to examine the type or properties of an object at runtime. Some programming languages possess this capability.

Introspection should not be confused with reflection, which goes a step further and is the ability for a program to manipulate the values, meta-data, properties and/or functions of an object at runtime. Some programming languages also possess that capability.