

UNIT-1

Preliminary Concepts

Background

- Frankly, we didn't have the vaguest idea how the thing [FORTRAN language and compiler] would work out in detail. ...We struck out simply to optimize the object program, the running time, because most people at that time believed you couldn't do that kind of thing. They believed that machine-coded programs would be so inefficient that it would be impractical for many applications.
- John Backus, unexpected successes are common – the browser is another example of an unexpected success

1.1 Reasons for Studying Concepts of Programming Languages- C01

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Overall advancement of computing

1.2 Programming Domains - C01

- Scientific applications
 - Large number of floating point computations
 - Fortran
- Business applications
 - Produce reports, use decimal numbers and characters
 - COBOL
- Artificial intelligence
 - Symbols rather than numbers manipulated
 - LISP
- Systems programming
 - Need efficiency because of continuous use
 - C
- Web Software
 - Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)

1.2 Language Evaluation Criteria – C01, C02

- Readability : the ease with which programs can be read and understood
- Writability : the ease with which a language can be used to create programs
- Reliability : conformance to specifications (i.e., performs to its specifications)
- Cost : the ultimate total cost

Readability

- Overall simplicity
 - A manageable set of features and constructs
 - Few feature multiplicity (means of doing the same operation)
 - Minimal operator overloading
- Orthogonality
 - A relatively small set of primitive constructs can be combined in a relatively small number of ways
 - Every possible combination is legal
- Control statements
 - The presence of well-known control structures (e.g., while statement)
- Data types and structures
 - The presence of adequate facilities for defining data structures
- Syntax considerations
 - Identifier forms: flexible composition
 - Special words and methods of forming compound statements
 - Form and meaning: self-descriptive constructs, meaningful keywords

Writability

- Simplicity and Orthogonality
 - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
 - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
 - A set of relatively convenient ways of specifying operations
 - Example: the inclusion of for statement in many modern languages

Reliability

- Type checking
 - Testing for type errors
- Exception handling
 - Intercept run-time errors and take corrective measures
- Aliasing
 - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
 - A language that does not support “natural” ways of expressing an algorithm will necessarily use “unnatural” approaches, and hence reduced reliability

Cost

- Training programmers to use language
 - Writing programs (closeness to particular applications)
 - Compiling programs
 - Executing programs
-
- Language implementation system: availability of free compilers
 - Reliability: poor reliability leads to high costs
 - Maintaining programs

Others

- Portability
- The ease with which programs can be moved from one implementation to another
- Generality
- The applicability to a wide range of applications
- Well-definedness
- The completeness and precision of the language's official definition

1.3 Influences on Language Design - CO3

- Computer Architecture
- Languages are developed around the prevalent computer architecture, known as the von Neumann architecture
- Programming Methodologies
- New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
- Data and programs stored in memory
- Memory is separate from CPU
- Instructions and data are piped from memory to CPU
- Basis for imperative languages
- Variables model memory cells
- Assignment statements model piping
- Iteration is efficient

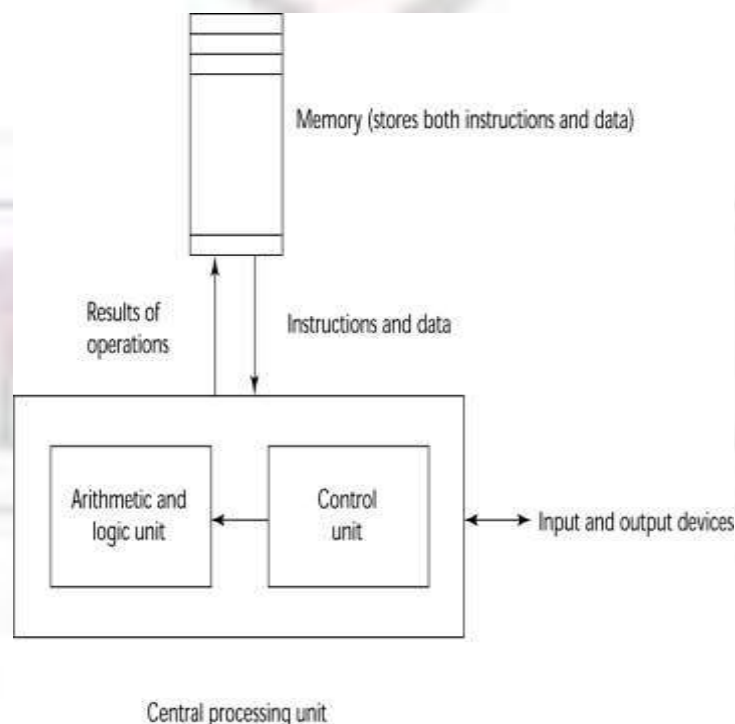


Figure 1.1 The von Neumann Computer Architecture

Programming Methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
- structured programming
- top-down design and step-wise refinement

- Late 1970s: Process-oriented to data-oriented
- data abstraction
- Middle 1980s: Object-oriented programming
- Data abstraction + inheritance + polymorphism

1.4 Language Categories – CO1

- Imperative
- Central features are variables, assignment statements, and iteration
- Examples: C, Pascal
- Functional
- Main means of making computations is by applying functions to given parameters
- Examples: LISP, Scheme
- Logic
- Rule-based (rules are specified in no particular order)
- Example: Prolog
- Object-oriented
- Data abstraction, inheritance, late binding
- Examples: Java, C++
- Markup
- New; not a programming per se, but used to specify the layout of information in Web documents
- Examples: XHTML, XML

Language Design Trade-Offs

- Reliability vs. cost of execution
- Conflicting criteria
- Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
- Another conflicting criteria
- Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
- Another conflicting criteria
- Example: C++ pointers are powerful and very flexible but not reliably used

1.5 Implementation Methods -CO2

- Compilation
- Programs are translated into machine language
- Pure Interpretation
- Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
- A compromise between compilers and pure interpreters

Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
- lexical analysis: converts characters in the source program into lexical units
- syntax analysis: transforms lexical units into parse trees which represent the syntactic structure of program
- Semantics analysis: generate intermediate code
- code generation: machine code is generated

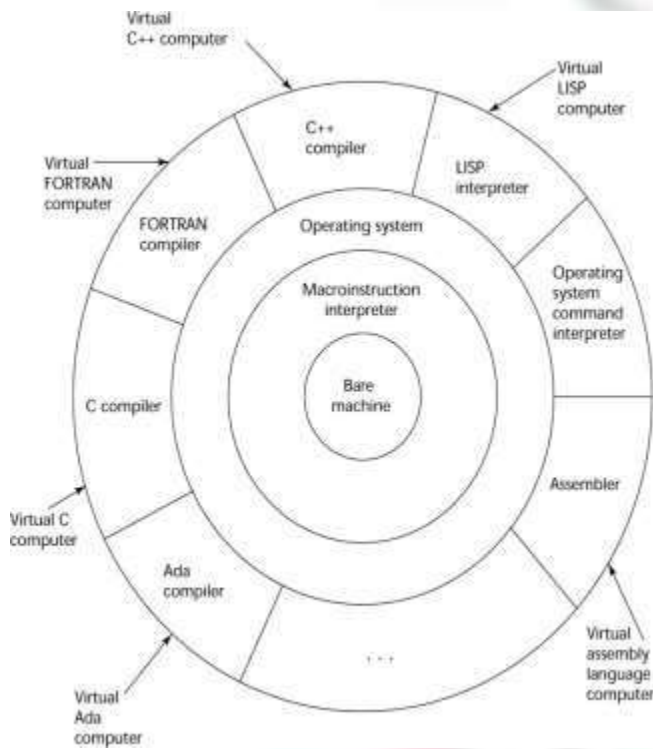


Figure 1.2 Layered View of Computer: The Operating system and language Implementation are layered over Machine Interface of a computer

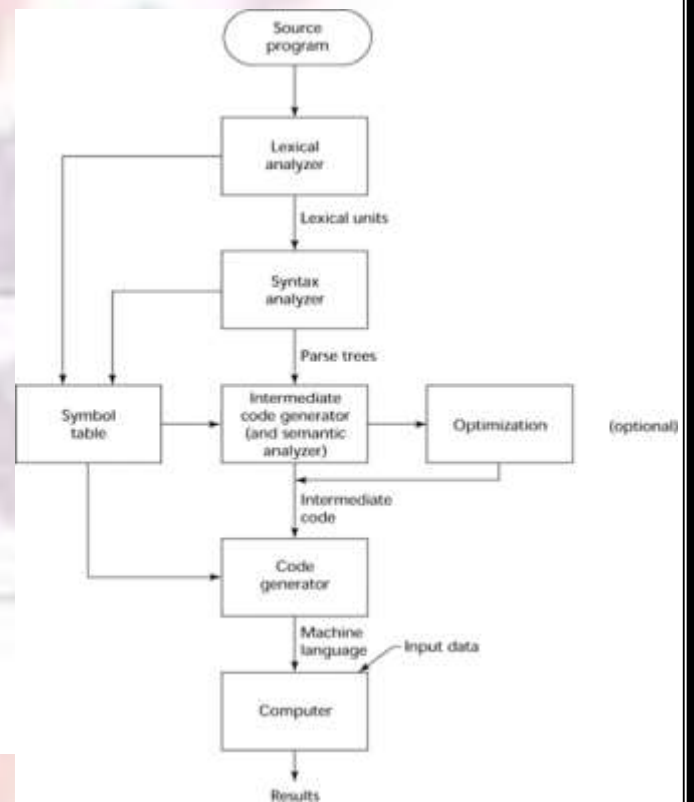


Figure 1.3: the compilation process

Additional Compilation

Terminologies

- Load module (executable image): the user and system code together

- Linking and loading: the process of collecting system program and linking them to user program

Execution of Machine Code

Fetch-execute-cycle (on a von Neumann architecture)

*initialize the program counter
repeat forever
 fetch the instruction pointed by
 the counter
 increment the counter
 decode the instruction
 execute the instruction
end repeat*

- **Von Neumann Bottleneck**
- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed a lot faster than the above connection speed; the connection speed thus results in a bottleneck
- Known as von Neumann bottleneck; it is the primary limiting factor in the speed of computers

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Becoming rare on high-level languages
- Significantly comeback with some latest web scripting languages (e.g., JavaScript)

Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
 - Perl programs are partially compiled to detect errors before interpretation
 - Initial implementations of Java were hybrid; the intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a runtime system (together, these are called Java Virtual Machine)

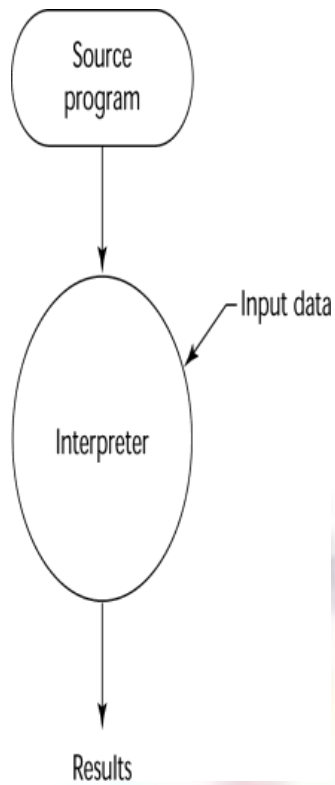


Figure 1.4 Pure Interpretation

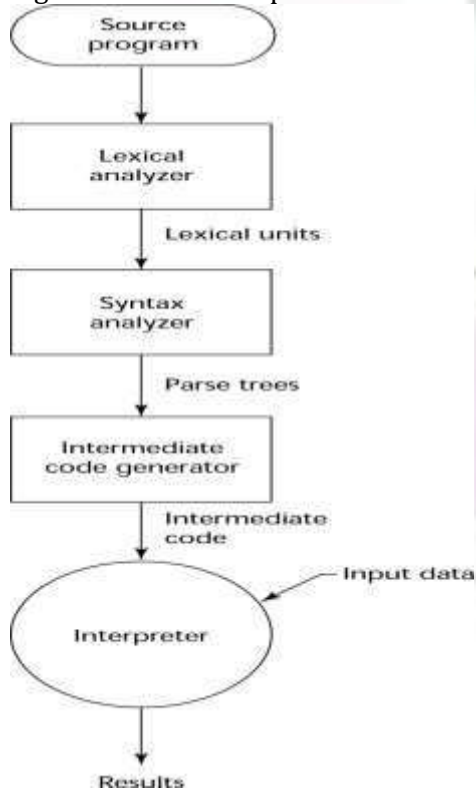


Figure 1.5 Hybrid Implementation

Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs

- .NET languages are implemented with a JIT system

1.6 Preprocessors - CO2

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
- expands #include, #define, and similar macros

1.7 Programming

Environments

Programming Environments

A programming environment is the collection of tools used in the development of software. This collection may consist:

- A file system,
- A text editor,
- A linker,
- A compiler,
- Integrated tools

These tools may be accessed through a uniform interface (GUI).

Some of the examples of programming environments are:

1. Microsoft Visual Studio .NET, which is a large collection of software development tools, used through a windows interface. It is used to develop software in following languages-
 1. C#,
 2. Visual Basic .NET,
 3. JScript (MS JavaScript version),
 4. J# (MS Java version),
 5. managed C++.
2. NetBeans
3. Turbo C, C++
4. Dreamweaver
5. Arduino, etc.

A programming environment is the collection of tools used in the development of software.

Few programming environments

Four of the most common IDEs are IntelliJ, Eclipse, NetBeans, and Visual Studio.

- The collection of tools used in software development

- **UNIX**

- An older operating system and tool collection

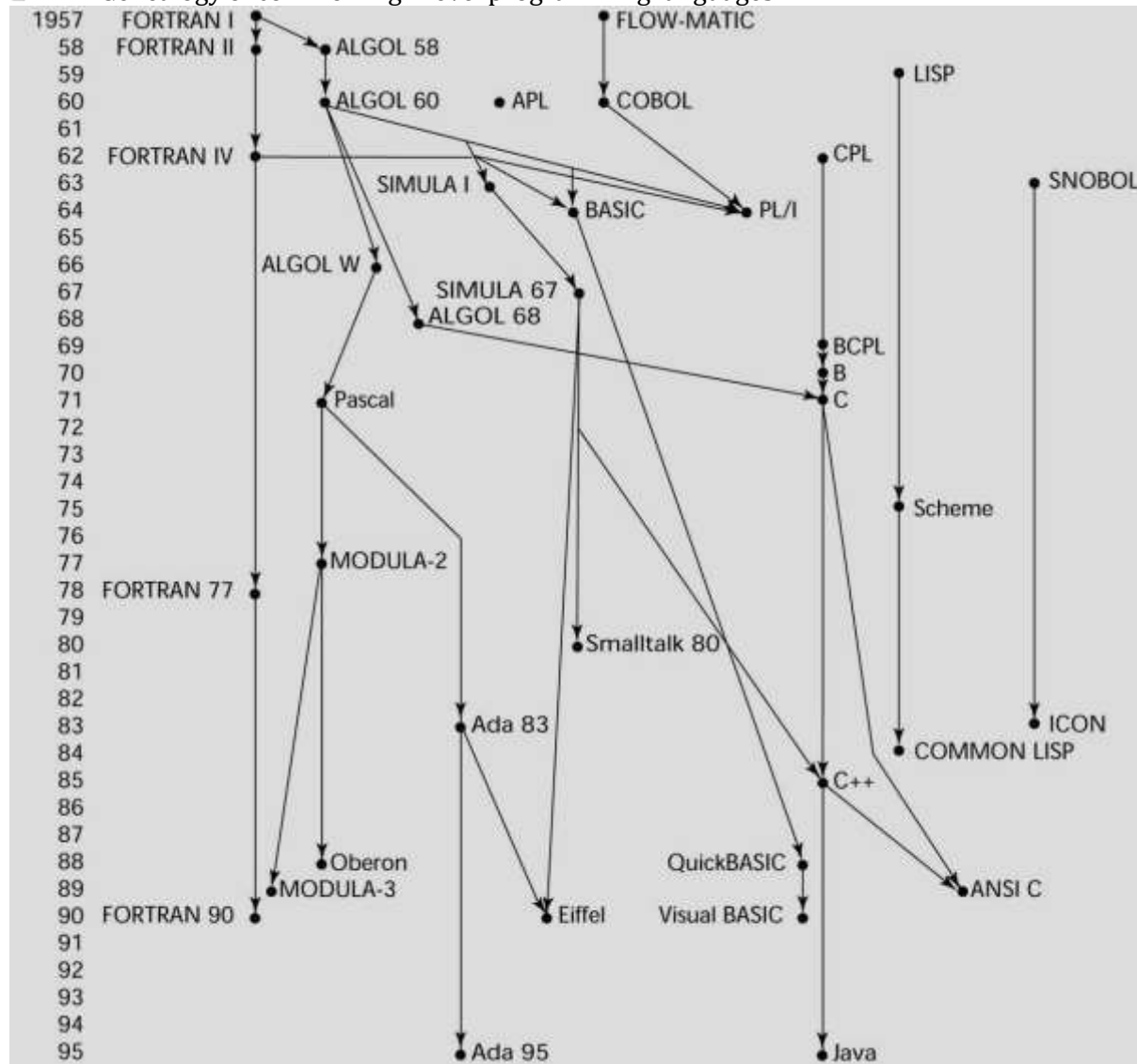
- **Borland JBuilder**

- An integrated development environment for Java

- **Microsoft Visual Studio.NET**

- A large, complex visual environment
- Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

7 Genealogy of common high-level programming languages



SYNTAX AND SEMANTICS

Introduction

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
 - Users of a language definition
 - Other language designers
 - Implementers
 - Programmers (the users of the language)
 -

Who must use language definitions?

- ☐ Other language designers
- Implementors
- Programmers (the users of the language)

1.3 The General Problem of Describing Syntax And Semantics - CO1

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (**e.g.**, *, sum, begin)
- A *token* is a category of lexemes (**e.g.**, identifier)
- **Languages Recognizers**
 - A recognition device reads input strings of the language and decides whether the input strings belong to the language
 - Example: syntax analysis part of a compiler
- **Languages Generators**
 - A device that generates sentences of a language
 - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator
- The design of a programming language can be divided into two parts: **syntax** and **semantics**.
- The syntax describes *how it looks like*.
- The semantics describes *what it should do*.
- There are many ways a program can be written with valid syntax but turn nonsensical when evaluated. These nonsensical evaluations are known as *runtime errors*.
- Semantics formally describes how programs should be evaluated. Programs that are well-formed according to its semantics do not get stuck.
- There are three main styles of describing semantics: operational, denotational, and axiomatic.

Operational semantics

Operational semantics uses the idea that languages are *abstract machines* and evaluation of a P program is a series of state transitions from an initial to a final state.

There are two major ways to write operational semantics: small-step or big-step.

Denotational semantics

Denotational semantics uses the idea that languages are mathematical objects.

Axiomatic semantics is an approach based on mathematical logic for proving the correctness of computer programs. It is closely related to Hoare logic. Axiomatic semantics define the meaning of a command in a program by describing its effect on assertions about the program state.

1.8 Formal Methods of Describing Syntax - CO1,C02

PROGRAMMING LANGUAGES (CS3112PE)

- Backus-Naur Form and Context-Free Grammars
- Most widely known method for describing programming language syntax
- Extended BNF
- Improves readability and writability of BNF
- Grammars and Recognizers

Backus-Naur Form and Context-Free Grammars

- Context-Free Grammars



- Developed by Noam Chomsky in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context-free languages

Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
 - Invented by John Backus to describe ALGOL 58
 - BNF is equivalent to context-free grammars
 - BNF is a metalanguage used to describe another language
 - In BNF, abstractions are used to represent classes of syntactic structures-- they act like syntactic variables (also called nonterminal symbols)

BNF Fundamentals

- Non-terminals: BNF abstractions
- Terminals: lexemes and tokens
- Grammar: a collection of rules
- Examples of BNF rules:

<ident_list> → identifier | identifier, <ident_list>

<if_stmt> → if <logic_expr> then <stmt>

BNF Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of terminal and nonterminal symbols
- A grammar is a finite nonempty set of rules
- An abstraction (or nonterminal symbol) can have more than one RHS

<stmt> → <single_stmt>

| begin <stmt_list> end

Describing Lists

- Syntactic lists are described using recursion

<ident_list> → ident

| ident, <ident_list>

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

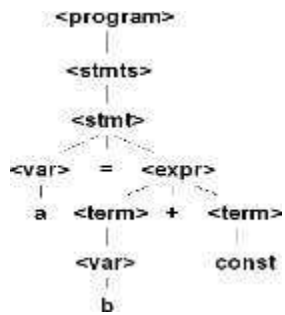
<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

Parse Tree

A hierarchical representation of a derivation

Figure 2.1 Parse Tree



An example derivation

- <program> ⇒ <stmts>
- ⇒ <stmt>
- ⇒ <var> = <expr>
- ⇒ a = <expr>
- ⇒ a = <term> + <term>
- ⇒ a = <var> + <term>
- ⇒ a = b + <term>
- ⇒ a = b + const

Derivation

- Every string of symbols in the derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Ambiguity in Grammars

- A grammar is ambiguous iff it generates a sentential form that has two or more distinct parse trees

An Unambiguous Expression Grammar

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

- <expr> → <expr> - <term> | <term>
- <term> → <term> / const | const

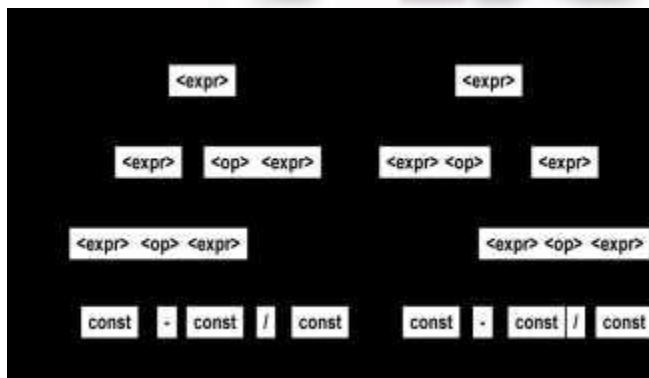


Figure 2.2 An Ambiguous Expression Grammar

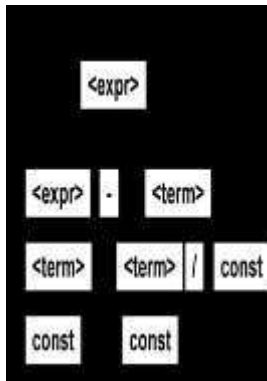


Figure 2.3 An Unambiguous Expression Grammar

Associativity of Operators

Operator associativity can also be indicated by a grammar

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \text{const}$ (ambiguous)

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \text{const} \mid \text{const}$ (unambiguous)

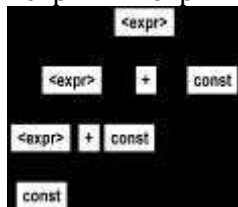


Figure 2.4 Parse Tree for Associativity operator

1.9 Extended Backus-Naur Form (EBNF) – CO2

- Optional parts are placed in brackets ([])
 $\langle \text{proc_call} \rangle \rightarrow \text{ident} [(\langle \text{expr_list} \rangle)]$
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle (+|-) \text{const}$
- Repetitions (0 or more) are placed inside braces ({ })
 $\langle \text{ident} \rangle \rightarrow \text{letter} \{\text{letter}|\text{digit}\}$

BNF and EBNF

- **BNF and EBNF**

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\mid \langle \text{expr} \rangle - \langle \text{term} \rangle$
 $\mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\mid \langle \text{term} \rangle / \langle \text{factor} \rangle$
 $\mid \langle \text{factor} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{(+ \mid -) \langle \text{term} \rangle\}$
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{(* \mid /) \langle \text{factor} \rangle\}$

1.10 Attribute Grammars – C01, C04

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Additions to CFGs to carry some semantic info along parse trees
- Primary value of attribute grammars (AGs):
 - Static semantics specification
 - Compiler design (static semantics checking)

Definition

- An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
 - For each grammar symbol x there is a set $A(x)$ of attribute values
 - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
 - Each rule has a (possibly empty) set of predicates to check for attribute consistency
 - Let $X_0 X_1 \dots X_n$ be a rule
 - Functions of the form $S(X_0) = f(A(X_1), \dots, A(X_n))$ define synthesized attributes
 - Functions of the form $I(X_j) = f(A(X_0), \dots, A(X_n))$, for $i \leq j \leq n$, define inherited attributes
 - Initially, there are intrinsic attributes on the leaves

Example

- Syntax
 - $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$
 - $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle \mid \langle \text{var} \rangle$
 - $\langle \text{var} \rangle \rightarrow A \mid B \mid C$
- `actual_type`: synthesized for `<var>` and `<expr>`
- `expected_type`: inherited for `<expr>`
- Syntax rule `<expr> → <var>[1] + <var>[2]`
- Semantic rules `<expr>.actual_type → <var>[1].actual_type`
- Predicate `<var>[1].actual_type == <var>[2].actual_type`
- `<expr>.expected_type == <expr>.actual_type`
- Syntax rule `<var> → id`
- Semantic rule `<var>.actual_type = lookup (<var>.string)`
- How are attribute values computed?
 - If all attributes were inherited, the tree could be decorated in top-down order.
 - If all attributes were synthesized, the tree could be decorated in bottom-up order.
 - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.
- `<expr>.expected_type = inherited from parent`
- `<var>[1].actual_type = lookup (A)`
- `<var>[2].actual_type = lookup (B)`
- `<var>[1].actual_type =? <var>[2].actual_type`
- `<expr>.actual_type = <var>[1].actual_type`
- `<expr>.actual_type =? <expr>.expected_type`

1.11 Describing the Meanings of Programs: Dynamic Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Operational Semantics
 - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the

meaning of the statement

- To use operational semantics for a high-level language, a virtual machine is needed
- A hardware pure interpreter would be too expensive
- A software pure interpreter also has problems:
 - The detailed characteristics of the particular computer would make actions difficult to understand
 - Such a semantic definition would be machine- dependent

Operational Semantics

- A better alternative: A complete computer simulation
- The process:
 - Build a translator (translates source code to the machine code of an idealized computer)
 - Build a simulator for the idealized computer
- Evaluation of operational semantics:
 - Good if used informally (language manuals, etc.)
 - Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.
- **Axiomatic Semantics**
 - Based on formal logic (predicate calculus)
 - Original purpose: formal program verification
 - Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
 - The expressions are called assertions

Axiomatic Semantics

- An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a postcondition
- A weakest precondition is the least restrictive precondition that will guarantee the postcondition
 - Pre-post form: {P} statement {Q}
 - An example: $a = b + 1 \{a > 1\}$
 - One possible precondition: $\{b > 10\}$
 - Weakest precondition: $\{b > 0\}$
 - Program proof process: The postcondition for the whole program is the desired result.

Work back through the program to the first statement. If the precondition on the first statement is the same as the program spec, the program is correct.

- An axiom for assignment statements ($x = E$):
 - $\{Qx \rightarrow E\} x = E \{Q\}$
 - An inference rule for sequences
 - For a sequence $S1;S2$:
 - $\{P1\} S1 \{P2\}$
 - $\{P2\} S2 \{P3\}$
 - An inference rule for logical pretest loops For the loop construct:
 - $\{P\} \text{ while } B \text{ do } S \text{ end } \{Q\}$ Characteristics of the loop invariant I must meet the following conditions:
 - $P \Rightarrow I$ (the loop invariant must be true initially)
 - $\{I\} B \{I\}$ (evaluation of the Boolean must not change the validity of I)
 - $\{I \text{ and } B\} S \{I\}$ (I is not changed by executing the body of the loop)
 - $(I \text{ and } (\text{not } B)) \Rightarrow Q$ (if I is true and B is false, Q is implied)
 - The loop terminates (this can be difficult to prove)
- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.

- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition.

Evaluation of Axiomatic Semantics:

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)
- The process of building a denotational spec for a language (not necessarily easy):
- Define a mathematical object for each language entity
- Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables
- The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions

- The state of a program is the values of all its current variables

$s = \{ \langle i1, v1 \rangle, \langle i2, v2 \rangle, \dots, \langle in, vn \rangle \}$

- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable

$\text{VARMAP}(ij, s) = v_j$

- Decimal Numbers

- The following denotational semantics description maps decimal numbers as strings of symbols into numeric values

$\langle \text{dec_num} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\mathbb{Z} \mid \langle \text{dec_num} \rangle (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

$\text{Mdec}('0') = 0, \text{Mdec}('1') = 1, \dots, \text{Mdec}('9') = 9$

$\text{Mdec}(\langle \text{dec_num} \rangle '0') = 10 * \text{Mdec}(\langle \text{dec_num} \rangle) \text{Mdec}(\langle \text{dec_num} \rangle '1') = 10 * \text{Mdec}(\langle \text{dec_num} \rangle) + 1$

...

$\text{Mdec}(\langle \text{dec_num} \rangle '9') = 10 * \text{Mdec}(\langle \text{dec_num} \rangle) + 9$

Expressions

- Map expressions onto \mathbb{Z} {error}
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

- Assignment Statements

- Maps state sets to state sets

- Logical Pretest Loops

- Maps state sets to state sets

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors

- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

- Recursion, when compared to iteration, is easier to describe with mathematical rigor
- Evaluation of denotational semantics
- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, they are of little use to language users

Summary

- BNF and context-free grammars are equivalent meta-languages
- Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
- Operation, axiomatic, denotational

-
-
-
-
-
-



UNIT-2

Names, Bindings & Scopes

1. Introduction
2. Names, Variable, concept of binding,
3. Scope, scope and life time
4. Referencing environment, Named constants

DATA TYPES

1. Introduction
2. primitive
3. character string type
4. user defined ordinal types, array
5. Associative array, record, union,
6. Tuple type, list types, pointer and reference types,
7. type checking, Strong typing
8. Type equivalence-design and implementation use related to these types.

Expressions and Statements

1. Arithmetic Expression, Over-loaded operators
2. Type conversion, relation & Boolean expression
3. Short circuit evolution, assignment statements, mixed mode assignments

Control Structure

1. Introduction, Selection Statements
2. Iterative Statements, Unconditional Branching,
3. Guarded Commands.

Names, Bindings and Scopes

Introduction

- Imperative languages are abstractions of von Neumann architecture
- Memory: stores both instructions and data
- Processor: provides operations for modifying the contents of memory
- Variables characterized by attributes
- Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

Design issues for names:

- Maximum length?
- Are connector characters allowed?
- Are names case sensitive?
- Are special words reserved words or keywords?

Name Forms

- ☒ A name is a string of characters used to identify some entity in a program.
- ☒ If too short, they cannot be connotative
- ☒ Language examples:
 - FORTRAN I: maximum 6
 - COBOL: maximum 30
 - FORTRAN 90 and ANSI C: maximum 31
 - Ada and Java: no limit, and all are significant
 - C++: no limit, but implementers often impose a length limitation because they do not want the symbol table in which identifiers are stored during compilation to be too large and also to simplify the maintenance of that table.
- ☒ Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and (_).
- ☒ Although the use of the _ was widely used in the 70s and 80s, that practice is far less popular.
- ☒ C-based languages (C, C++, Java, and C#), replaced the _ by the “camel” notation, as in myStack.
- ☒ Prior to Fortran 90, the following two names are equivalent:

Sum Of Salaries // names could have embedded spaces SumOfSalaries // which were ignored

- Case sensitivity
- Disadvantage: readability (names that look alike are different)
- worse in C++ and Java because predefined names are mixed case (e.g.

IndexOutOfBoundsException)

- In C, however, exclusive use of lowercase for names.
- C, C++, and Java names are case sensitive ☒ rose, Rose, ROSE are distinct names “What about Readability”

Special words

- An aid to readability; used to delimit or separate statement clauses
- A keyword is a word that is special only in certain contexts.
- Ex: Fortran

Real Apple // Real is a data type followed with a name, therefore Real is a keyword
 Real = 3.4 // Real is a variable name

- **Disadvantage:** poor readability. Compilers and users must recognize the difference.
- A reserved word is a special word that cannot be used as a user-defined name.
- As a language design choice, reserved words are better than keywords.
- Ex: In Fortran, one could have the statements

Integer Real // keyword "Integer" and variable "Real" Real Integer// keyword "Real" and variable "Integer"

Variables

- A variable is an abstraction of a memory cell(s).
- Variables can be characterized as a sextuple of attributes:
- Name
- Address
- Value
- Type
- Lifetime
- Scope

Name

- Not all variables have names: Anonymous, heap-dynamic variables

Address

- The memory address with which it is associated
- A variable name may have different addresses at different places and at different times during execution.

// sum in sub1 and sub2

- A variable may have different addresses at different times during execution. If a subprogram has a local var that is allocated from the run time stack when the subprogram is called, different calls may result in that var having different addresses.

// sum in sub1

- The address of a variable is sometimes called its l-value because that is what is required when a variable appears in the left side of an assignment statement.

Aliases

- If two variable names can be used to access the same memory location, they are called aliases
- Aliases are created via pointers, reference variables, C and C++ unions.
- Aliases are harmful to readability (program readers must remember all of them)

Type

- Determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision.
- For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

Value

- The value of a variable is the contents of the memory cell or cells associated with the variable.

- Abstract memory cell - the physical cell or collection of cells associated with a variable.
- A variable's value is sometimes called its r-value because that is what is required when a variable appears in the right side of an assignment statement.

The Concept of Binding

- The **l-value** of a variable is its address.
- The **r-value** of a variable is its value.
- A **binding** is an association, such as between an attribute and an entity, or between an operation and a symbol.
- **Binding time** is the time at which a binding takes place.
- Possible binding times:
 - Language design time: bind operator symbols to operations.
 - For example, the asterisk symbol (*) is bound to the multiplication operation.
 - Language implementation time:
 - A data type such as int in C is bound to a range of possible values.
 - Compile time: bind a variable to a particular data type at compile time.
 - Load time: bind a variable to a memory cell (ex. C static variables)
 - Runtime: bind a nonstatic local variable to a memory cell.

Binding of Attributes to Variables

- A binding is static if it first occurs before run time and remains unchanged throughout program execution.
- A binding is dynamic if it first occurs during execution or can change during execution of the program.

Type Bindings

- If static, the type may be specified by either an explicit or an implicit declaration.

Variable Declarations

- An explicit declaration is a program statement used for declaring the types of variables.
- An **implicit declaration** is a **default** mechanism for specifying types of variables (the first appearance of the variable in the program.)
- Both explicit and implicit declarations create static bindings to types.
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations.
- EX:
 - In Fortran, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention:
I, J, K, L, M, or N or their lowercase versions is implicitly declared to be Integer type; otherwise, it is implicitly declared as Real type.
 - **Advantage:** writability.
 - **Disadvantage:** reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.
 - In Fortran, vars that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.
 - Less trouble with Perl: Names that begin with \$ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure.
 - In this scenario, the names @apple and %apple are unrelated.
 - In **C and C++**, one must distinguish between declarations and definitions.
 - **Declarations** specify types and other attributes but do not cause allocation of storage. Provides the type of a var defined external to a function that is used in the function.
 - Definitions specify attributes and cause storage allocation.

Dynamic Type Binding (JavaScript and PHP)

- Specified through an assignment statement
- Ex, JavaScript
list = [2, 4.33, 6, 8]; ❏ single-dimensioned array list = 47; ❏ scalar variable

- Advantage: flexibility (generic program units)
- Disadvantages:
 - High cost (dynamic type checking and interpretation)
 - Dynamic type bindings must be implemented using pure interpreter not compilers.
 - Pure interpretation typically takes at least ten times as long as to execute equivalent machine code.

Type error detection by the compiler is difficult because any variable can be assigned a value of any type.

- Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.
- Ex:
 - $i, x \rightarrow \text{Integer}$
 - $y \rightarrow \text{floating-point array}$
 - $i = x \rightarrow \text{what the user meant to type}$ $i = y \rightarrow \text{what the user typed instead}$
 - No error is detected by the compiler or run-time system. i is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

Type Inference (ML, Miranda, and Haskell)

- Rather than by assignment statement, types are determined from the context of the reference.

• Ex:

$\text{fun circumf}(r) = 3.14159 * r * r;$

The argument and functional value are inferred to be real.

$\text{fun times10}(x) = 10 * x;$

The argument and functional value are inferred to be int.

Storage Bindings & Lifetime

- **Allocation** - getting a cell from some pool of available cells.
- **Deallocation** - putting a cell back into the pool.
- The lifetime of a variable is the time during which it is bound to a particular memory cell. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.
- Categories of variables by lifetimes: static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic

Static Variables:

- bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
- e.g. all FORTRAN 77 variables, C static variables.
- **Advantages:**
 - **Efficiency:** (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocating and deallocating vars.
 - History-sensitive: have vars retain their values between separate executions of the subprogram.

Disadvantage:

- Storage cannot be shared among variables.
- Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

Stack-dynamic Variables:

- Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.

- Ex:
 - The variable declarations that appear at the beginning of a Java method are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.
- Stack-dynamic variables are allocated from the run-time stack.
- If scalar, all attributes except address are statically bound.
- Ex:
 - Local variables in C subprograms and Java methods.
- Advantages:
 - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
 - In the absence of recursion it conserves storage b/c all subprograms share the same memory space for their locals.
- Disadvantages:
 - Overhead of allocation and deallocation.
 - Subprograms cannot be history sensitive.
 - Inefficient references (indirect addressing) is required b/c the place in the stack where a particular var will reside can only be determined during execution.
- In Java, C++, and C#, variables defined in methods are by default stack-dynamic.

Explicit Heap-dynamic Variables:

- Nameless memory cells that are allocated and deallocated by explicit directives “run-time instructions”, specified by the programmer, which take effect during execution.
- These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- The heap is a collection of storage cells whose organization is highly disorganized b/c of the unpredictability of its use.
- e.g. dynamic objects in C++ (via new and delete)

```
int *intnode;
intnode = new int; // allocates an int cell
delete intnode; // deallocates the cell to which
// intnode points
```

- An explicit heap-dynamic variable of int type is created by the new operator.
- This operator can be referenced through the pointer, intnode.
- The var is deallocated by the delete operator.
- Java, all data except the primitive scalars are objects.
- Java objects are explicitly heap-dynamic and are accessed through reference variables.
- Java uses **implicit garbage collection**.
- Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.
- Advantage:
 - Provides for dynamic storage management.
- Disadvantage:
 - Inefficient “Cost of allocation and deallocation” and unreliable “difficulty of using pointer and reference variables correctly”

Implicit Heap-dynamic Variables:

- Bound to heap storage only when they are assigned value. Allocation and deallocation caused by assignment statements.
- All their attributes are bound every time they are assigned.
- e.g. all variables in APL; all strings and arrays in Perl and JavaScript.
- Advantage:
 - Flexibility allowing generic code to be written.
- Disadvantages:

- Inefficient, because all attributes are dynamic “run-time.”
- Loss of error detection by the compiler.

Scope

- The scope of a var is the range of statements in which the var is visible.
- A var is visible in a statement if it can be referenced in that statement.
- Local var is local in a program unit or block if it is declared there.
- Non-local var of a program unit or block are those that are visible within the program unit or block but are not declared there.

Static Scope

- Binding names to non-local vars is called static scoping.
- There are two categories of static scoped languages:
 - ☒ Nested Subprograms.
 - ☒ Subprograms that can't be nested.
- Ada, and JavaScript allow nested subprogram, but the C-based languages do not.
- When a compiler for static-scoped language finds a reference to a var, the attributes of the var are determined by finding the statement in which it was declared.
- Ex: Suppose a reference is made to a var x in subprogram Sub1. The correct declaration is found by first searching the declarations of subprogram Sub1.
- If no declaration is found for the var there, the search continues in the declarations of the subprogram that declared subprogram Sub1, which is called its static parent.

If a declaration of x is not found there, the search continues to the next larger enclosing unit (the unit that declared Sub1's parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success. ☒ an undeclared var error has been detected.

- The static parent of subprogram Sub1, and its static parent, and so forth up to and including the main program, are called the static ancestors of Sub1.

Ex: Ada procedure:

```

- Procedure Big is
- X : Integer;
- Procedure Sub1 is
- Begin                                -- of Sub1
- ...X...
- end;                                -- of Sub1
- Procedure Sub2 isX Integer;
- Begin                                -- of Sub2
- ...X...
- end;                                -- of Sub2
- Begin                                -- of Big
- ...
- end;                                -- of Big
    
```

- Under static scoping, the reference to the var X in Sub1 is to the X declared in the procedure Big.
- This is true b/c the search for X begins in the procedure in which the reference occurs, Sub1, but no declaration for X is found there.
- The search thus continues in the static parent of Sub1, Big, where the declaration of X is found.

- Ex: Skeletal C#

```

void sub()
{
    int count;
    ...
    while (...)
    {
    
```



```

int count; count ++;
...
}
...
}

```

- The reference to count in the while loop is to that loop's local count. The count of sub is hidden from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a var with the same name in a larger enclosing scope.
- C++ and Ada allow access to these "hidden" variables
- ☒ In Ada: Main.X
- ☒ In C++: class_name::name

Blocks

- Allows a section of code to have its own local vars whose scope is minimized.
- Such vars are stack dynamic, so they have their storage allocated when the section is entered and deallocated when the section is exited.
- From ALGOL 60:
- Ex:

C and C++:

```

for (...)
{
int index;
...
}

```

Ada:

```

declare LCL : FLOAT; begin
...
End

```

Dynamic Scope

- ☒ The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic.
- ☒ Based on calling sequences of program units, not their textual layout (temporal versus spatial) and thus the scope is determined at run time.
- ☒ References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- ☒ Ex:

```

Procedure Big is X : Integer; Procedure Big is
  X : Integer;
  Procedure Sub1 is
    Begin      -- of Sub1
    ...X...
    end;      -- of Sub1
  Procedure Sub2 isX
    Integer;
    Begin      -- of Sub2
    ...X...
    end;      -- of Sub2
  Begin      -- of Big
  ...
  end;      -- of Big

```

- ☒ Big calls Sub1
- o The dynamic parent of Sub1 is Big. The reference is to the X in Big.
- ☒ Big calls Sub2 and Sub2 calls Sub1

o The search proceeds from the local procedure, Sub1, to its caller, Sub2, where a declaration of X is found.

☞ Note that if static scoping was used, in either calling sequence the reference to X in Sub1 would be to Big's X.

Scope and Lifetime

☞ Ex:

```
void printhead()
{
...
} /* end of printhead */
void compute()
{
int sum;
... printhead();
} /* end of compute */
```

☞ The scope of sum is contained within compute.

☞ The lifetime of sum extends over the time during which printhead executes.

☞ Whatever storage location sum is bound to before the call to printhead, that binding will continue during and after the execution of printhead.

Referencing environment

☞ It is the collection of all names that are visible in the statement.

- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes.

- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to non-local vars in both static and dynamic scoped languages.

- A subprogram is active if its execution has begun but has not yet terminated.

- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms.

Ex, Ada, static-scoped language procedure Example is A, B : Integer;
 procedure Sub1 is A, B : Integer;

```
...
procedure Sub1 is
☞   X, Y : Integer;
   begin    -- of Sub1
   ...
   end    -- of Sub1
procedure Sub2 is
   X : Integer;
   ...
   procedure Sub3 is
     X : Integer;
     begin  -- of Sub3
     ...
     end;  -- of Sub3
   begin  -- of Sub2
   ...
   end;  { Sub2}
begin
...
end;    {Example}
```

Point Referencing Environment

- 1 X and Y of Sub1, A & B of Example
 - 2 X of Sub3, (X of Sub2 is hidden), A and B of Example
 - 3 X of Sub2, A and B of Example
 - 4 A and B of Example
- ☒ Ex, dynamic-scoped language
- ☒ Consider the following program; assume that the only function calls are the following: main calls sub2, which calls sub1

```

void sub1( )
{
  int a, b;
  ...
} /* end of sub1 */
void sub2( )
{
  int b, c;
  ...
  sub1;
} /* end of sub2 */
void main ( )
{
  int c, d;
  ...
  sub2( );
} /* end of main */

```

← 1

← 2

← 3

The referencing environments of the indicates program points are as follows:

Point Referencing Environment

- 1 a and b of sub1, c of sub2, d of main
- 2 b and c of sub2, d of main
- 3 c and d of main

Named Constants

☒ It is a var that is bound to a value only at the time it is bound to storage; its value can't be change by assignment or by an input statement.

☒ Ex, Java

```
final int LEN = 100;
```

☒ Advantages: readability and modifiability

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization.
- Initialization is often done on the declaration statement.
- Ex, Java

```
int sum = 0;
```

Data Types and Variables

Introduction

- A data type defines a collection of data objects and a set of predefined operations on those objects
 - A descriptor is the collection of the attributes of a variable
 - An object represents an instance of a user-defined (abstract data) type
 - One design issue for all data types: What operations are defined and how are they specified?
- 2.1 Primitive Data Types – CO2**

- Almost all programming languages provide a set of primitive data types
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- Java's signed integer sizes: byte, short, int, long

Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., float and double; sometimes more)
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754

Complex

- Some languages support a complex type, e.g., Fortran and Python
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):
(7 + 3j), where 7 is the real part and 3 is the imaginary part

Decimal

- For business applications (money)
- Essential to COBOL
- C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- Advantage: accuracy
- Disadvantages: limited range, wastes memory

Boolean

- Simplest of all
- Range of values: two elements, one for "true" and one for "false"
- Could be implemented as bits, but often as bytes
- Advantage: readability

Character

- Stored as numeric coding
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode
- Includes characters from most natural languages
- Originally used in Java
- C# and JavaScript also support Unicode

Character String Types

- Values are sequences of characters
- Design issues:
 - Is it a primitive type or just a special kind of array?
 - Should the length of strings be static or dynamic?

Operations

- Typical operations:
 - Assignment and copying
 - Comparison (=, >, etc.)
 - Catenation
 - Substring reference
 - Pattern matching

Character String Type in Certain Languages

- C and C++
 - Not primitive

- Use char arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
- Primitive
- Many operations, including elaborate pattern matching
- Fortran and Python
- Primitive type with assignment and several operations
- Java
- Primitive via the String class
- Perl, JavaScript, Ruby, and PHP
- Provide built-in pattern matching, using regular expressions

Character String Length Options

- Static: COBOL, Java’s String class
- Limited Dynamic Length: C and C++
- In these languages, a special character is used to indicate the end of a string’s characters, rather than maintaining the length
- Dynamic (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Evaluation

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide—why not have them?
- Dynamic length is nice, but is it worth the expense?

Implementation

- Static length: compile-time descriptor
- Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)
- Dynamic length: need run-time descriptor; allocation/de-allocation is the biggest implementation problem

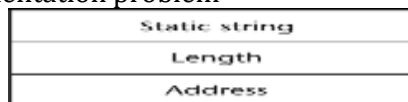


Figure 3.1 Compile-Time Descriptor

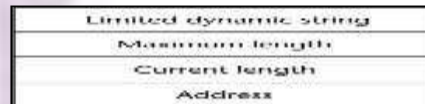


Figure 3.2 Run-Time Descriptors

2.1 User-Defined Ordinal Types - CO2

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of primitive ordinal types in Java
 - integer
 - char
 - boolean

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C# example
enum days {mon, tue, wed, thu, fri, sat, sun};
- Design issues
 - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
 - Are enumeration values coerced to integer?
 - Any other type coerced to an enumeration type?

Evaluation of Enumerated Type

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
- Operations (don’t allow colors to be added)

- No enumeration variable can be assigned a value outside its defined range
- Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Subrange Types

- An ordered contiguous subsequence of an ordinal type
 - Example: 12..18 is a subrange of integer type
 - Ada's design
- type Days is (mon, tue, wed, thu, fri, sat, sun); subtype Weekdays is Days range mon..fri; subtype Index is Integer range 1..100;

Day1: Days; Day2: Weekday; Day2 := Day1;

Subrange Evaluation

- Aid to readability
- Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
- Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementation

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Array Types

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

Array Indexing

- Indexing (or subscripting) is a mapping from indices to elements
array_name (index_value_list) → an element
- Index Syntax
- FORTRAN, PL/I, Ada use parentheses
- Ada explicitly uses parentheses to show uniformity between array references and function calls because both are mappings
- Most other languages use brackets

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- Index range checking
- C, C++, Perl, and Fortran do not specify range checking
- Java, ML, C# specify range checking
- In Ada, the default is to require range checking, but it can be turned off

Subscript Binding and Array Categories

- Static: subscript ranges are statically bound and storage allocation is static (before run-

time)

- Advantage: efficiency (no dynamic allocation)
- Fixed stack-dynamic: subscript ranges are statically bound, but the allocation is done at declaration time
- Advantage: space efficiency
- Stack-dynamic: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
- Advantage: flexibility (the size of an array need not be known until the array is to be used)
- Fixed heap-dynamic: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)
- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
- Advantage: flexibility (arrays can grow or shrink during program execution)
- C and C++ arrays that include static modifier are static
- C and C++ arrays without static modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class ArrayList that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialization

- Some language allow initialization at the time of storage allocation
- C, C++, Java, C# example
int list [] = {4, 5, 7, 83}
- Character strings in C and C++
char name [] = "freddie";
- Arrays of strings in C and C++
char *names [] = {"Bob", "Jake", "Joe"};
- Java initialization of String objects
String[] names = {"Bob", "Jake", "Joe"};

Heterogeneous Arrays

- A heterogeneous array is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)
- Ada allows array assignment but also catenation
- Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array catenation
- Fortran provides elemental operations because they are between pairs of array elements
- For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
- Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)

Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism

- Slices are only useful in languages that have array operations

Slice Examples

– Fortran 95

Integer, Dimension (10) :: Vector Integer, Dimension (3, 3) :: Mat

Integer, Dimension (3, 3) :: Cube

Vector (3:6) is a four element array

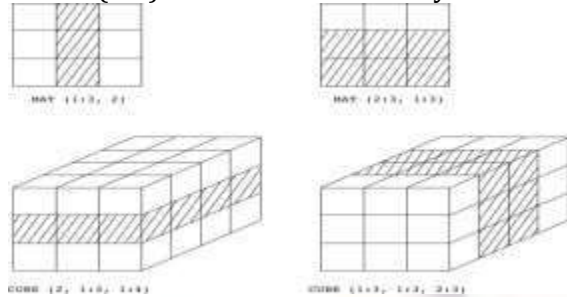


Figure 3.3 Slices Examples in Fortran 95

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:
 $address(list[k]) = address(list[lower_bound]) + ((k - lower_bound) * element_size)$

Accessing Multi-dimensioned Arrays

- Two common ways:
 - Row major order (by rows) – used in most languages
 - Column major order (by columns) – used in Fortran

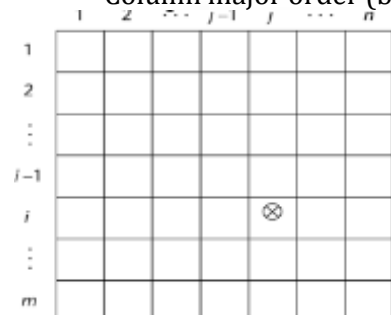


Figure 3.4 Locating an Element in a Multi-dimensioned Array

Compile-Time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Figure 3.5 Single-Dimensioned Array

Associative Arrays

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range <i>n</i>
Address

Figure 3.6 Multi-Dimensioned Array

- An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys
 - User-defined keys must be stored
- Design issues:
 - What is the form of references to elements?
 - Is the size static or dynamic?

Associative Arrays in Perl

- Names begin with %; literals are delimited by parentheses
`%hi_temps = ("Mon" => 77, "Tue" => 79, -Wed|| => 65, ...);`
- Subscripting is done using braces and keys
`$hi_temps{"Wed"} = 83;`
- Elements can be removed with delete
`delete $hi_temps{"Tue"};`

2.2 Record Types – CO2

- A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
 - What is the syntactic form of references to the field?
 - Are elliptical references allowed?

Definition of Records in COBOL

- COBOL uses level numbers to show nested records; others use recursive definition
- ```

01 EMP-REC.
02 EMP-NAME.
05 FIRST PIC X(20).
05 MID PIC X(10).
05 LAST PIC X(20).
02 HOURLY-RATE PIC 99V99.

```

**Definition of Records in Ada**

- Record structures are indicated in an orthogonal way

```

type Emp_Rec_Type is record
 First: String (1..20);
 Mid: String (1..10);
 Last: String (1..20);
 Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;

```

**References to Records**

- Record field references
  - COBOL

```
field_name OF record_name_1 OF ... OF record_name_n
```

  - Others (dot notation)

```
record_name_1.record_name_2 record_name_n.field_name
```
- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL

FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

**Operations on Records**

- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING
  - Copies a field of the source record to the corresponding field in the target record

**Evaluation and Comparison to Arrays**

- Records are used when collection of data values is heterogeneous
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

**Unions Types**

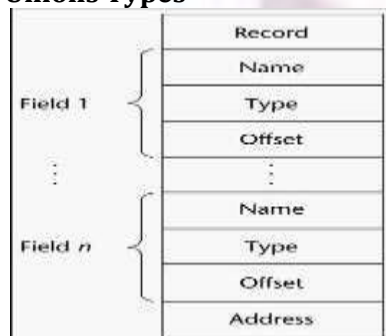


Figure 3.7 Implementation of Record Type

- A union is a type whose variables are allowed to store different type values at different times during execution
- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

**Discriminated vs. Free Unions**

- Fortran, C, and C++ provide union constructs in which there is no language support for

type checking; the union in these languages is called free union

- Type checking of unions require that each union include a type indicator called a discriminant
- Supported by Ada

### Ada Union Types

*type Shape is (Circle, Triangle, Rectangle);*

*type Colors is (Red, Green, Blue);*

*type Figure (Form: Shape) is record*

*Filled: Boolean;*

*Color: Colors;*

*case Form is*

*when Circle => Diameter: Float;*

*when Triangle => Leftside, Rightside: Integer;*

*Angle: Float;*

*when Rectangle => Side1, Side2: Integer; end case;*

*end record;*

Evaluation of Unions

- Free unions are unsafe
- Do not allow type checking
- Java and C# do not support unions

Ada Union Type Illustrated

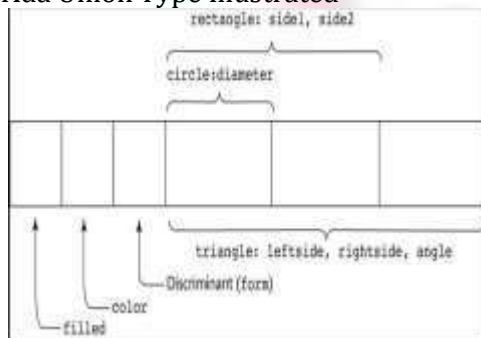


Figure 3.8 A Discriminated Union of Three Shape Variables

- Reflective of growing concerns for safety in programming language
- Ada's discriminated unions are safe

### Pointer and Reference Types

- A pointer type variable has a range of values that consists of memory addresses and a special value, nil
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a heap)

### Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

### Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value

## PROGRAMMING LANGUAGES (CS3112PE)

- Dereferencing can be explicit or implicit
- C++ uses an explicit operation via \*

`j = *ptr`

sets j to the value located at ptr

Pointer Assignment Illustration



### Pointer Assignment Illustration

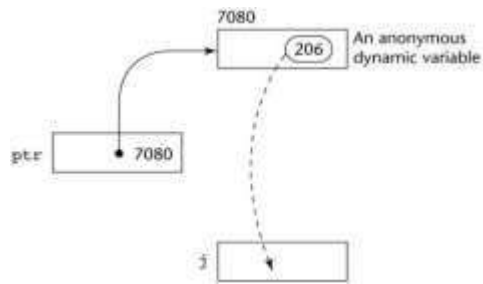


Figure 3.9 The assignment operation  $j = *ptr$

#### Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called garbage)
- Pointer p1 is set to point to a newly created heap-dynamic variable
- Pointer p1 is later set to point to another newly created heap-dynamic variable
- The process of losing heap-dynamic variables is called memory leakage

#### Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada (possible with UNCHECKED\_DEALLOCATION)

#### Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when or where it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (void \*)

void \* can point to any type and can be type checked (cannot be de-referenced)

#### Pointer Arithmetic in C and C++

float stuff[100]; float \*p;

p = stuff;

\*(p+5) is equivalent to stuff[5] and p[5]

\*(p+i) is equivalent to stuff[i] and p[i]

#### Reference Types

- C++ includes a special kind of pointer type called a reference type that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

#### Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers are like goto's--they widen the range of cells that can be accessed by a variable
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

#### Representations of Pointers

- Large computers use single values
- Intel microprocessors use segment and offset

#### Dangling Pointer Problem

- Tombstone: extra heap cell that is a pointer to the heap-dynamic variable



- The actual pointer variable points only at tombstones
- When heap-dynamic variable de-allocated, tombstone remains but set to nil
- Costly in time and space
- Locks-and-keys: Pointer values are represented as (key, address) pairs
- Heap-dynamic variables are represented as variable plus cell for integer lock value
- When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

**Heap Management**

- A very complex run-time process
- Single-size cells vs. variable-size cells
- Two approaches to reclaim garbage
- Reference counters (eager approach): reclamation is gradual
- Mark-sweep (lazy approach): reclamation occurs when the list of variable space becomes empty

**Reference Counter**

- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
- Disadvantages: space required, execution time required, complications for cells connected circularly
- Advantage: it is intrinsically incremental, so significant delays in the application execution are avoided

**Mark-Sweep**

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
- Every heap cell has an extra bit used by collection algorithm
- All cells initially set to garbage
- All pointers traced into heap, and reachable cells marked as not garbage
- All garbage cells returned to list of available cells
- Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary markswep algorithms avoid this by doing it more often—called incremental markswep

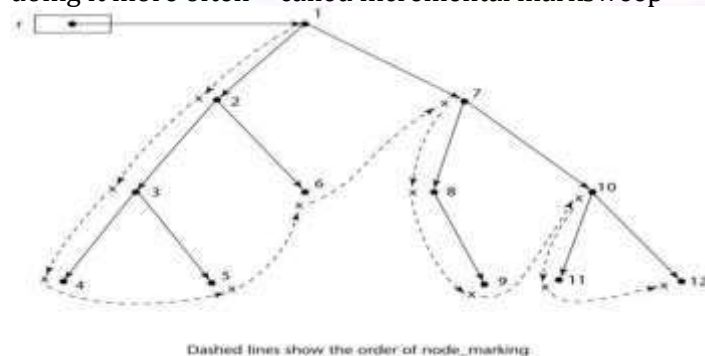


Figure 3.10 Marking Algorithm

**Variable-Size Cells**

- All the difficulties of single-size cells plus more
- Required by most programming languages
- If mark-sweep is used, additional problems occur
- The initial setting of the indicators of all cells in the heap is difficult
- The marking process is nontrivial
- Maintaining the list of available space is another source of overhead

**2.3 Type Checking - CO3**

- Generalize the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type. This automatic conversion is called as coercion.
- A type error is the application of an operator to an operand of an inappropriate type
- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- Def: A programming language is strongly typed if type errors are always detected

### Type Compatibility

- Our concern is primarily for structured types
- Def: Name type compatibility means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)
- Structure type compatibility means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
- Consider the problem of two structured types:
  - Are two record types compatible if they are structurally the same but use different field names?
  - Are two array types compatible if they are the same except that the subscripts are different? (e.g., [1..10] and [0..9])
  - Are two enumeration types compatible if their components are spelled differently?
  - With structural type compatibility, you cannot differentiate between types of the same structure (e.g., different units of speed, both float)
- Language examples:
  - Pascal: usually structure, but in some cases name is used (formal parameters)
  - C: structure, except for records
  - Ada: restricted form of name
  - o Derived types allow types with the same structure to be different
  - o Anonymous types are all unique, even in:
    - A, B : array (1..10) of INTEGER:

### 2.4 Strong Typing - CO2

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
  - FORTRAN 77 is not: parameters, EQUIVALENCE
  - Pascal is not: variant records
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked

- Ada is, almost (UNCHECKED CONVERSION is loophole) (Java is similar)
- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

### Named Constants

- Def: A named constant is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called manifest constants) or dynamic
- Languages:
  - Pascal: literals only
  - FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind

### Variable Initialization

- Def: The binding of a variable to a value at the time it is bound to storage is called initialization
- Initialization is often done on the declaration statement e.g., Java `int sum = 0`

### Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management
- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors

### Tuple

A tuple is a compound data type having a fixed number of terms. Each term in a tuple is known as an element. The number of elements is the size of the tuple.

**A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.)**

Tuple. Tuples are used to store multiple items in a single variable. Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage. A tuple is a collection which is ordered and unchangeable.

In computer science, tuples come in many forms. Most typed functional programming languages implement tuples directly as product types,[1] tightly associated with algebraic data types, pattern matching, and destructuring assignment.[2] Many programming languages offer an alternative to tuples, known as record types, featuring unordered elements accessed by label.[3] A few programming languages combine ordered tuple product types and unordered record types into a single construct, as in C structs and Haskell records. Relational databases may formally identify their rows (records) as tuples.

**Advantages of Tuple**

Tuples offer the following advantages –

- Tuples are fixed size in nature i.e. we can't add/delete elements to/from a tuple.
- We can search any element in a tuple.
- Tuples are faster than lists, because they have a constant set of values.
- Tuples can be used as dictionary keys, because they contain immutable values like strings, numbers, etc.

**List Types**

• A list is a number of items in an ordered or unordered structure. A list can be used for a number of things like storing items or deleting and adding items. But for the programmer to perform the different tasks for the list, the program must have enough memory to keep up with changes done to the list. List is the most versatile data type available in functional programming languages used to store a collection of similar data items. The concept is similar to arrays in object-oriented programming. List items can be written in a square bracket separated by commas. The way to writing data into a list varies from language to language.

- There is different sort of lists which are linear list and linked list. Also, the list can be referred to as an abstract data type.
- **Linear List** - A static abstract data type. The amount of data does not change at run time.
- **Linked List** - Dynamic Abstract Data Type. Uses pointers to vary memory used at run time.
- List is not a data type in Java/C/C++, but we have alternative ways to create a list in Java, i.e., by using ArrayList and LinkedList.

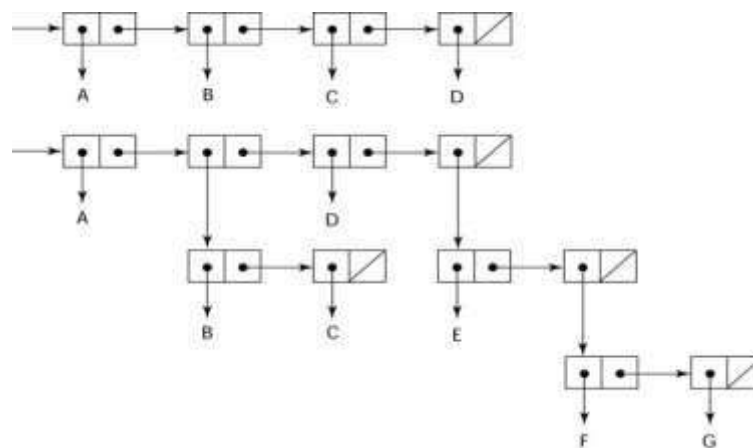
For example, in LISP, (A B C D)

Nested lists have the same form, so we could have (A (B C) D) is composed of 3 elements

In which A & D are atoms & (B C) are sub-list

- OPERATIONS IN LIST ARE: add, remove/ delete, update, searching, sorting.
- 

**Representation of Two LISP Lists**



Representing the lists (A B C D)

and (A (B C) D (E (F G)))

**TYPE EQUIVALENCE**

**Name Type Equivalence**

Name type equivalence means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name

Easy to implement but highly restrictive:

- Subranges of integer types are not equivalent with integer types
- Formal parameters must be the same type as their corresponding actual parameters

**Structure Type Equivalence**

Structure type equivalence means that two variables have equivalent types if their types have identical structures

More flexible, but harder to implement

Consider the problem of two structured types:

- Are two record types equivalent if they are structurally the same but use different field names?
- Are two array types equivalent if they are the same except that the subscripts are different? (e.g. [1...10] and [0...9])
- Are two enumeration types equivalent if their components are spelled differently?
- With structural type equivalence, you cannot differentiate between types of the same structure (e.g., different units of speed, both float)

**Expressions and Statements & Control Structures****Introduction**

- Expressions are the fundamental means of specifying computations in a programming language.
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- Essence of imperative languages is dominant role of assignment statements Arithmetic Expressions.
- Arithmetic evaluation was one of the motivations for the development of the first programming languages.

**2.6 Arithmetic Expressions – CO2, CO3**

Arithmetic Expressions consist of operators, operands, parentheses and function calls.

**Design Issues**

Design issues for arithmetic expressions

- Operator precedence rules?
- Operator associativity rules?
- Order of operand evaluation?
- Operand evaluation side effects?
- Operator overloading?
- Type mixing in expressions?

**Operators**

- A unary operator has one operand.
- A binary operator has two operands.
- A ternary operator has three operands.

**Operator Precedence Rules**



The operator precedence rules for expression evaluation define the order in which adjacent operators of different precedence levels are evaluated.

Typical precedence levels:

- parentheses
- unary operators
- \*\* (if the language supports it)
- \*, /
- +, -

### Operator Associativity Rule

The operator associativity rules for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated.

Typical associativity rules:

- Left to right, except \*\* (Ruby and Fortran), which is right to left
- Sometimes unary operators associate right to left (e.g., in FORTRAN)
- APL is different; all operators have equal precedence and all operators associate right to left.
- Precedence and associativity rules can be overridden with parentheses.

### Conditional Expressions

Conditional Expressions (ternary operator ?:) available in C-based languages. An example: (C, C++)

```
average = (count == 0)? 0 : sum/count
```

Evaluates as if written like

```
if (count == 0)
```

```
average = 0
```

```
else
```

```
average = sum /count
```

### Operand Evaluation

#### Order

Operand evaluation order as follows.

- Variables: fetch the value from memory.
- Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction.
- Parenthesized expressions: evaluate all operands and operators first.
- The most interesting case is when an operand is a function call.

**Potentials for Side Effects** Functional side effects: when a function changes a two-way parameter or a non-local variable

Problem with functional side effects: When a function referenced in an expression alters another operand of the expression;

e.g., for a parameter change:

```
a = 10;
```

```
/* assume that fun changes its parameter */
```

```
b = a + fun(a);
```

Two possible solutions to the functional side effects problem: Write the language definition to disallow functional side effects.

- No two-way parameters in functions
- No non-local references in functions

Advantage: it works!

Disadvantages: inflexibility of one-way parameters and lack of non-local references

- Write the language definition to demand that operand evaluation order be fixed.



Disadvantage: limits some compiler optimizations

- Java requires that operands appear to be evaluated in left-to-right order.

### **Overloaded Operators**

Use of an operator for more than one purpose is called operator overloading.

- Some are common (e.g., + for int and float)
- Some are potential trouble (e.g., \* in C and C++)

Problems:

- Loss of compiler error detection (omission of an operand should be a detectable error)
- Some loss of readability
- Can be avoided by introduction of new symbols (e.g., Pascal's div for integer division)
- C++, Ada, Fortran 95, and C# allow user-defined overloaded operators

Potential problems:

- Users can define nonsense operations.
- Readability may suffer, even when the operators make sense.

### **Type Conversions**

A narrowing conversion is one that converts an object to a type that cannot include all of the values of the original type.

e.g., float to int

A widening conversion is one in which an object is converted to a type that can include at least approximations to all of the values of the original type.

e.g., int to float

### **Mixed Mode**

A mixed-mode expression is one that has operands of different types A coercion is an implicit type conversion

### **Disadvantage of coercions:**

- They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions.
- In Ada, there are virtually no coercions in expressions Explicit Type Conversions called as casting in C-based languages. Examples:-

C: (int)angle, Ada: Float (Sum)

- Note that Ada's syntax is similar to that of function calls

### **Errors in Expressions causes**

- Inherent limitations of arithmetic e.g., division by zero
- Limitations of computer arithmetic e.g., overflow
- Often ignored by the run-time system

### **Relational and Boolean Expressions**

#### **Relational Expressions:**

- Use relational operators and operands of various types
- Evaluate to some Boolean representation
- Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)
- JavaScript and PHP have two additional relational operator, === and !==
- Similar to their cousins, == and !=, except that they do not coerce their operands

#### **Boolean Expressions:**

Operands are Boolean and the result is Boolean Example operators

|       | FORTRAN 77 | FORTRAN 90 | C   | Ada  |
|-------|------------|------------|-----|------|
| .AND. | and        | &&         | and |      |
| .OR.  | or         |            | or  |      |
| .NOT. | not        | !          | not |      |
| xor   | ---        | ---        | --- | ---- |

- No Boolean Type in C
- C89 has no Boolean type--it uses int type with 0 for false and nonzero for true
- One odd characteristic of C's expressions:  $a < b < c$  is a legal expression, but the result is not what you might expect:
  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., c)

### 2.7 Short Circuit Evaluation - CO3

An expression in which the result is determined without evaluating all of the operands and/or operators

Example:  $(13 * a) * (b / 13 - 1)$

If 'a' is zero, there is no need to evaluate  $(b / 13 - 1)$

Problem with non-short-circuit evaluation

index = 1;

while (index <= length) && (LIST[index] != value) index++;

- When index=length, LIST [index] will cause an indexing problem (assuming LIST has length -1 elements)
- C, C++ and Java: use short-circuit evaluation for the usual Boolean operators (&& and ||), but also provide bitwise Boolean operators that are not short circuit (& and |)
- Ada: programmer can specify either (short-circuit is specified with and then and or else)
- Short-circuit evaluation exposes the potential problem of side effects in expressions e.g.,  $(a > b) || (b++ / 3)$

### 2.8 Assignment Statements CO3

The general syntax: <target\_var> <assign\_operator> <expression> The assignment operator

- '=' in FORTRAN, BASIC, the C-based languages
- ':=' in ALGOL, Pascal, Ada

-Equal '=' can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)

Conditional Targets (Perl)

$(\$flag ? \$total : \$subtotal) = 0$

Which is equivalent to

if ( $\$flag$ ){ $\$total = 0$ } else { $\$subtotal = 0$ }

#### Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment.
- Introduced in ALGOL; adopted by C Ex: 'a = a + b' is written as 'a += b'

#### Unary Assignment Operators

Unary assignment operators in C-based languages combine increment and decrement operations with assignment. For example

sum += ++count (count incremented, added to sum) sum += count++ (count incremented, added to sum) count++ (count incremented)

-count++ (count incremented then negated)

#### Assignment as an Expression

In C, C++, and Java, the assignment statement produces a result and can be used as operands.

while ((ch = getchar()) != EOF){...}

ch = getchar() is carried out; the result (assigned to ch) is used as a conditional value for the while statement

#### List Assignments

Perl and Ruby support list assignments

e.g., ( $\$first, \$second, \$third$ ) = (20, 30, 40);

### Mixed-Mode Assignment

Assignment statements can also be mixed-mode, for example

```
int a, b; float c;
```

```
c = a / b;
```

- In Fortran, C, and C++, any numeric type value can be assigned to any numeric type variable.
- In Java, only widening assignment coercions are done.
- In Ada, there is no assignment coercion.

### 2.9 Control Structures - CO3

A control structure is a control statement and the statements whose execution it controls.

Levels of Control Flow

- Within expressions
- Among program units
- Among program statements

Control Statements: Evolution

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the 1960s about the issue

One important result: It was proven that all algorithms represented by flowcharts can be coded with only two-way selection and pretest logical loops

### 2.10 Selection Statements - CO3

A selection statement provides the means of choosing between two or more paths of execution.

Two general categories:

- Two-way selectors
- Multiple-way selectors

#### *Two-Way Selection Statements*

General form as follows...

```
if control_expression
then clause
```

```
else clause
```

Design Issues:

- What is the form and type of the control expression?
- How are the then and else clauses specified?
- How should the meaning of nested selectors be specified?

The Control Expression

- If the 'then' reserved word or some other syntactic marker is not used to introduce the 'then' clause, the control expression is placed in parentheses.
- In C89, C99, Python, and C++, the control expression can be arithmetic.
- In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean.

Clause Form

- In many contemporary languages, the then and else clauses can be single statements or compound statements
- In Perl, all clauses must be delimited by braces (they must be compound)
- In Fortran 95, Ada, and Ruby, clauses are statement sequences
- Python uses indentation to define clauses

```
if x > y :
```

```
x = y
```

```
print "case 1"
```

Nesting Selectors:Java example

```
if (sum == 0) if (count == 0) result = 0;
```

```
else result = 1;
```

- Which if gets the else?
- Java's static semantics rule: else matches with the nearest if Nesting Selectors
- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) { if (count == 0) result = 0;} else result = 1;
```

- The above solution is used in C, C++, and C#
- Perl requires that all then and else clauses to be compound
- Statement sequences as clauses: Ruby

```
if sum == 0 then if count == 0 then result = 0
else result = 1 end
end
```

#### -Python

```
if sum == 0 :
if count == 0 :
result = 0 else :
result = 1
```

### Multiple-Way Selection Statements

Allow the selection of one of any number of statements or statement groups Design Issues:

- What is the form and type of the control expression?
- How are the selectable segments specified?
- Is execution flow through the structure restricted to include just a single selectable segment?
- How are case values specified?
- What is done about unrepresented expression values?

### Multiple-Way Selection: Examples

- C, C++, and Java

```
switch (expression) {
case const_expr_1: stmt_1;
```

...

```
case const_expr_n: stmt_n; [default: stmt_n+1]}
```

- Design choices for C's **switch** statement
- Control expression can be only an integer type
- Selectable segments can be statement sequences, blocks, or compound statements
- Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)

- **default** clause is for unrepresented values (if there is no default, the whole statement does nothing)

- C#

– Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment

– Each selectable segment must end with an unconditional branch (goto or break)

- Ada

```
case expression is
when choice list => stmt_sequence;
```

...

```
when choice list => stmt_sequence; when others => stmt_sequence;] end case;
```

More reliable than C's switch (once a stmt\_sequence execution is completed, control is passed to the first statement after the case statement)

- Ada design choices:

1. Expression can be any ordinal type
2. Segments can be single or compound

3. Only one segment can be executed per execution of the construct
4. Unrepresented values are not allowed
  - Constant List Forms:
    1. A list of constants
    2. Can include:
      - Subranges
      - Boolean OR operators (|)

### Multiple-Way Selection Using if

Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

```
if count < 10 :
bag1 = True
elif count < 100 :
bag2 = True
elif count < 1000 :
bag3 = True
```

### Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

### Counter-Controlled Loops

A counting iterative statement has a loop variable, and a means of specifying the initial and terminal, and stepsize values

Design Issues:

- What are the type and scope of the loop variable?
- What is the value of the loop variable at loop termination?
- Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?

### Iterative Statements: Examples

FORTRAN 95 syntax

DO label var = start, finish [, stepsize] Step size can be any value but zero Parameters can be expressions

Design choices:

1. Loop variable must be INTEGER
2. Loop variable always has its last value
3. The loop variable cannot be changed in the loop, but the parameters can; because they are evaluated only once, it does not affect loop control
4. Loop parameters are evaluated only once

- FORTRAN 95 : a second form:

[name:] Do variable = initial, terminal [,stepsize]

...

End Do [name]

- Cannot branch into either of Fortran's Do statements

- Ada

for var in [reverse] discrete\_range loop ... end loop

- Design choices:

- Type of the loop variable is that of the discrete range (A discrete range is a sub-range of an integer or enumeration type).

- Loop variable does not exist outside the loop

- The loop variable cannot be changed in the loop, but the discrete range can; it does not



affect loop control

- The discrete range is evaluated just once
- Cannot branch into the loop body
- C-based languages

for ([expr\_1] ; [expr\_2] ; [expr\_3]) statement

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas
- The value of a multiple-statement expression is the value of the last statement in the expression

- If the second expression is absent, it is an infinite loop

• Design choices:

- There is no explicit loop variable
- Everything can be changed in the loop
- The first expression is evaluated once, but the other two are evaluated with each iteration

• C++ differs from C in two ways:

- The control expression can also be Boolean
- The initial expression can include variable definitions (scope is from the definition to the end of the loop body)

• Java and C#

- Differs from C++ in that the control expression must be Boolean

- Iterative Statements: Logically-Controlled Loops

- Repetition control is based on a Boolean expression

• Design issues:

- Pretest or posttest?

- Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

• Iterative Statements: Logically-Controlled Loops: Examples

C and C++ have both pretest and posttest forms, in which the control expression can be arithmetic:

while (ctrl\_expr) do loop body loop body while (ctrl\_expr)

- Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no goto)

• Iterative Statements: Logically-Controlled Loops: Examples

- Ada has a pretest version, but no posttest

• FORTRAN 95 has neither

- Perl and Ruby have two pretest logical loops, while and until. Perl also has two posttest loops

### **Unconditional Branching:** User-Located Loop Control Mechanisms

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)

• Simple design for single loops (e.g., break)

• Design issues for nested loops

- Should the conditional be part of the exit?

- Should control be transferable out of more than one loop? User-Located Loop Control Mechanisms break and continue

• C, C++, Python, Ruby, and C# have unconditional unlabeled exits (break)

• Java and Perl have unconditional labeled exits (break in Java, last in Perl)

• C, C++, and Python have an unlabeled control statement, continue, that skips the remainder of the current iteration, but does not exit the loop

• Java and Perl have labeled versions of continue

Iterative Statements: Iteration Based on Data Structures



- Number of elements of in a data structure control loop iteration
- Control mechanism is a call to an iterator function that returns the next element in some chosen order, if there is one; else loop is terminate

• C's for can be used to build a user-defined iterator:

```
for (p=root; p!=NULL; traverse(p)){ }
```

- C#'s foreach statement iterates on the elements of arrays and other collections:

```
Strings[] = strList = {"Bob", "Carol", "Ted"}; foreach (Strings name in strList) Console.WriteLine ("Name: {0}", name);
```

The notation {0} indicates the position in the string to be displayed

- Perl has a built-in iterator for arrays and hashes, foreach Unconditional Branching
- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960's and 1970's
- Well-known mechanism: goto statement
- Major concern: Readability
- Some languages do not support goto statement (e.g., Java)
- C# offers goto statement (can be used in switch statements)
- Loop exit statements are restricted and somewhat camouflaged goto's

### **Guarded Commands**

- Designed by Dijkstra
- Purpose: to support a new programming methodology that supported verification (correctness) during development
- Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada)
- Basic Idea: if the order of evaluation is not important, the program should not specify one

### **Selection Guarded Command**

- Form

```
if <Boolean exp> -> <statement> [] <Boolean exp> -> <statement>
```

...

```
[] <Boolean exp> -> <statement> fi
```

Semantics: when construct is reached,

- Evaluate all Boolean expressions
- If more than one are true, choose one non-deterministically
- If none are true, it is a runtime error

### **Loop Guarded Command**

- Form

```
do <Boolean> -> <statement> [] <Boolean> -> <statement>
```

...

```
[] <Boolean> -> <statement> Od
```

### **Semantics: for each iteration**

- Evaluate all Boolean expressions
- If more than one are true, choose one non-deterministically; then start loop again
- If none are true, exit loop

### **Guarded Commands: Rationale**

- Connection between control statements and program verification is intimate
- Verification is impossible with goto statements
- Verification is possible with only selection and logical pretest loops
- Verification is relatively simple with only guarded commands

### **Summary**

- Expressions

- Operator precedence and associativity
- Operator overloading
- Mixed-type expressions
- Various forms of assignment
- Variety of statement-level structures
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability
- Functional and logic programming languages are quite different control structures



## UNIT-3

### Subprograms and Blocks

#### Introduction

- Two fundamental abstraction facilities
  - Process abstraction
- Emphasized from early days
  - Data abstraction
- Emphasized in the 1980s

#### 3.1 Fundamentals of Subprograms - CO4

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates

#### Basic Definitions

- A subprogram definition describes the interface to and the actions of the subprogram abstraction
- A subprogram call is an explicit request that the subprogram be executed
- A subprogram header is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
  - The parameter profile (aka signature) of a subprogram is the number, order, and types of its parameters
  - The protocol is a subprogram's parameter profile and, if it is a function, its return type
  - Function declarations in C and C++ are often called prototypes
  - A subprogram declaration provides the protocol, but not the body, of the subprogram
  - A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram
    - An actual parameter represents a value or address used in the subprogram call statement

#### Actual/Formal Parameter Correspondence

- Positional
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - Parameters can appear in any order

#### Formal Parameter Default Values

- In certain languages (e.g., C++, Ada), formal parameters can have default values (if not actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated

- C# methods can accept a variable number of parameters as long as they are of the same type

### Procedures and Functions

- There are two categories of subprograms
- Procedures are collection of statements that define parameterized computations
- Functions structurally resemble procedures but are semantically modeled on mathematical functions
- They are expected to produce no side effects
- In practice, program functions have side effects

### 3.2 Design Issues for Subprograms -CO3

- What parameter passing methods are provided?
- Are parameter types checked?
- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- Can subprograms be overloaded?
- Can subprogram be generic?

### Scope and Lifetime

- The scope of a variable is the range of statements over which it is visible
- The nonlocal variables of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables
- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a static variable in a C or C++ function

#### Static scope

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent
- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
- In Ada: **unit.name**
- In C++: **class\_name::name**
- Blocks
- A method of creating static scopes inside program units--from ALGOL 60
- Examples:

C and C++: for (...)

```
{ int index;
```

```
...
```

```
}
```

Ada: declare LCL : FLOAT; begin

```
...
```

```
end
```

- Evaluation of Static Scoping

- Consider the example:

Assume MAIN calls A and B

A calls C and D B calls A and E

- Suppose the spec is changed so that D must now access some data in B
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access
- Overall: static scoping often encourages many globals

### **Dynamic Scope**

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

### Scope Example

MAIN

- declaration of x SUB1

- declaration of x -

...

call SUB2

... SUB2

...

- reference to x -

...

...

call SUB1

...

- Static scoping
  - Reference to x is to MAIN's x
- Dynamic scoping
  - Reference to x is to SUB1's x
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability

### **Local Referencing Environments**

- Def: The referencing environment of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms
- Local variables can be stack-dynamic (bound to storage)
  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages
    - Allocation/de-allocation, initialization time
- Indirect addressing
- Subprograms cannot be history sensitive
- Local variables can be static
  - More efficient (no indirection)
  - No run-time overhead

### 3.3 Parameter Passing Methods – C03

- Ways in which parameters are transmitted to and/or from called subprograms
- Pass-by-value
- Pass-by-result
- Pass-by-value-result
- Pass-by-reference
- Pass-by-name

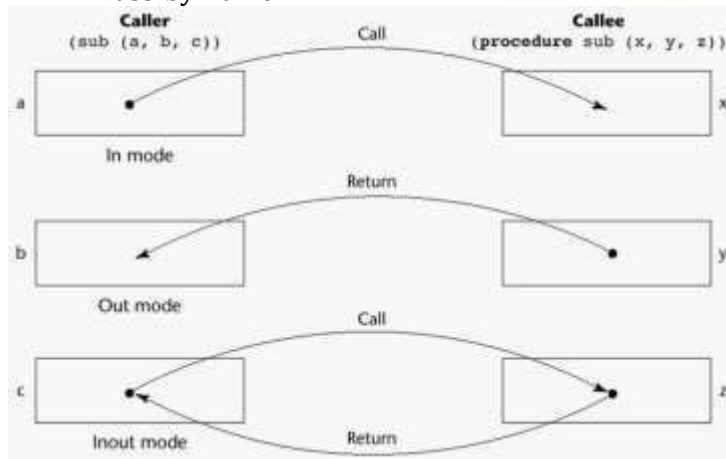


Figure 5.1 Models of Parameter Passing

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - When copies are used, additional storage is required
  - Storage and copy operations can be costly

#### Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller
  - Require extra storage location and copy operation
- Potential problem: sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1

#### Pass-by-Value-Result (Inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

#### Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for un-wanted side effects
  - Un-wanted aliases (access broadened)



### Pass-by-Name (Inout Mode)

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding

### Implementing Parameter-Passing

#### Methods

- In most language parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack
- A subtle but fatal error can occur with pass-by-reference and pass-by-value result: a formal parameter corresponding to a constant can mistakenly be changed

### Parameter Passing Methods of Major Languages

- Fortran
  - Always used the inout semantics model
  - Before Fortran 77: pass-by-reference
  - Fortran 77 and later: scalar variables are often passed by value-result
- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All parameters are passed are passed by value
  - Object parameters are passed by reference
- Ada
  - Three semantics modes of parameter transmission: in, out, in out; in is the default mode
  - Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; in out parameters can be referenced and assigned
- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref
- PHP: very similar to C#
- Perl: all actual parameters are implicitly placed in a predefined array named @\_

### Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal, FORTRAN 90, Java, and Ada: it is always required
- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking

### Multidimensional Arrays as Parameters

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function

### Multidimensional Arrays as Parameters: C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

### Multidimensional Arrays as Parameters: Pascal and Ada

- Pascal
  - Not a problem; declared size is part of the array's type

- Ada
- Constrained arrays - like Pascal
- Unconstrained arrays - declared size is part of the object declaration

#### **Multidimensional Arrays as Parameters: Fortran**

- Formal parameter that are arrays have a declaration after the header
- For single-dimension arrays, the subscript is irrelevant
- For multi-dimensional arrays, the subscripts allow the storage-mapping function

#### **Multidimensional Arrays as Parameters: Java and C#**

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (length in Java, Length in C#) that is set to the length of the array when the array object is created

#### Design Considerations for Parameter Passing

- Two important considerations
- Efficiency
- One-way or two-way data transfer
- But the above considerations are in conflict
- Good programming suggest limited access to variables, which means one way whenever possible
- But pass-by-reference is more efficient to pass structures of significant size

#### **3.4 Parameters Subprograms as parameters - CO3**

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
- Are parameter types checked?
- What is the correct referencing environment for a subprogram that was sent as a parameter?

#### **Parameters that are Subprogram Names: Parameter Type Checking**

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed; parameters can be type checked
- FORTRAN 95 type checks

- Later versions of Pascal and
- Ada does not allow subprogram parameters; a similar alternative is provided via Ada's generic facility

#### **Parameters that are Subprogram Names: Referencing Environment**

- Shallow binding: The environment of the call statement that enacts the passed subprogram
- Deep binding: The environment of the definition of the passed subprogram
- Ad hoc binding: The environment of the call statement that passed the subprogram

#### **Calling Subprograms Indirectly**

- Usually when there are several possible subprograms to be called and the correct one on a particular run of the program is not known until execution (e.g., event handling and GUIs).
  - In C and C++, such calls are made through function pointers
  - In C#, method pointers are implemented as objects called delegates
  - A delegate declaration: `public delegate int Change (int x);`
  - This delegate type, named Change, can be instantiated with any method that takes an int parameter and returns an int value
- A method: `static int fun1(int x) {...}`  
Instantiate: `Change chgfun1 = new Change(fun1);`  
Can be called with: `chgfun1(12);`  
- A delegate can store more than one address, which is called a multicast delegate

**3.5 Overloaded Subprograms – C03**

- An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

**3.6 Generic Subprograms – C03**

- A generic or polymorphic subprogram takes parameters of different types on different activations
  - Overloaded subprograms provide ad hoc polymorphism
  - A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides parametric polymorphism

**Examples of parametric polymorphism: C++**

```
template <class Type>
Type max(Type first, Type second) { return first > second ? first : second;
}
```

- The above template can be instantiated for any type for which operator > is defined
 

```
int max (int first, int second) { return first > second? first : second;
}
```

**Design Issues for Functions**

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Ada allows any type
  - Java and C# do not have functions but methods can have any type

**User-Defined Overloaded Operators**

- Operators can be overloaded in Ada and C++
- An Ada example
 

```
Function "*" (A,B: in Vec_Type): return Integer is Sum: Integer := 0;
begin
for Index in A'range loop
Sum := Sum + A(Index) * B(Index) end loop
return sum; end "*";
...
c = a * b; -- a, b, and c are of type Vec_Type
```

**CLOSURES**

- Closures (i.e., subroutine + non-local environment) are needed only when subroutines can be nested
  - Object-oriented languages without nested subroutines can use objects to implement a form of closure
    - > a method plays the role of the subroutine
    - > instance variables provide the non-local environment
  - Objects playing the role of a function + non-local environment are called object closures or function objects

- Ad-hoc syntax in some languages
- > In C++ an object of a class that overrides operator () can be called with functional syntax

### 3.7 Co-Routines - C03

- A coroutine is a subprogram that has multiple entries and controls them itself
- Also called symmetric control: caller and called coroutines are on a more equal basis
- A coroutine call is named a resume
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasi-concurrent execution of program units (the coroutines); their execution is interleaved, but not overlapped

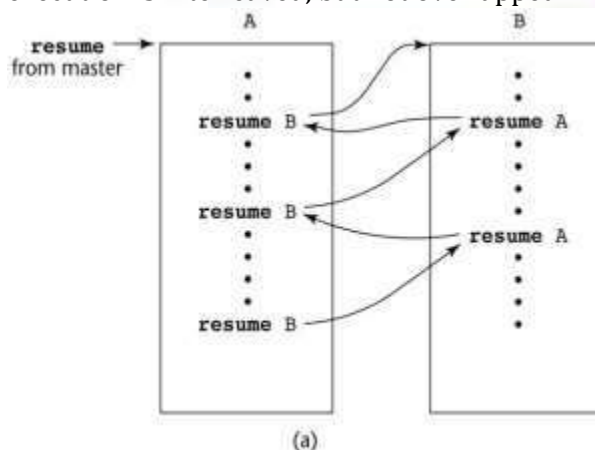


Figure 5.2 Possible Execution Controls

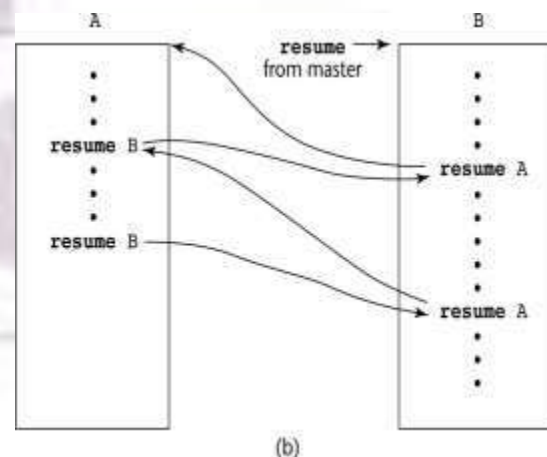


Figure 5.3 Possible Execution Controls

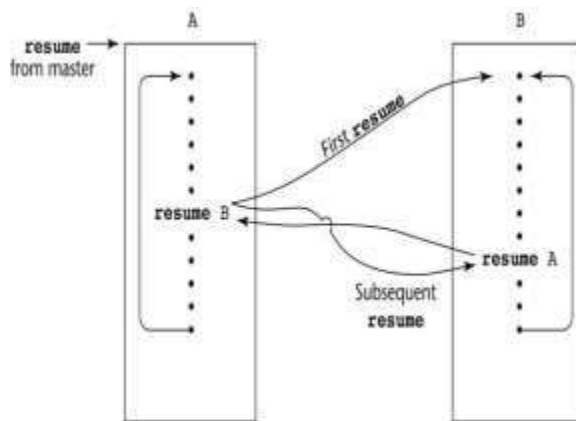


Figure 5.4 Possible Execution Controls with Loops

**Summary**

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and in out mode
- Some languages allow operator overloading
- Subprograms can be generic
- A co-routine is a special subprogram with multiple entries

## Abstract Data Types

**3.8 The Concept of Abstraction - CO3**

- An abstraction is a view or representation of an entity that includes only the most significant attributes
- The concept of abstraction is fundamental in programming (and computer science)
- Nearly all programming languages support process abstraction with subprograms
- Nearly all programming languages designed since 1980 support data abstraction

**3.9 Introduction to Data Abstraction - CO3**

- An Abstract Data Type is a user-defined data type that satisfies the following two conditions:
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit
  - The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition

**Advantages of Data Abstraction**

- Advantage of the first condition
  - Program organization, modifiability (everything associated with a data structure is together), and separate compilation



- Advantage the second condition
- Reliability--by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting user code

Language Requirements for ADTs:

- A syntactic unit in which to encapsulate the type definition
- A method of making type names and subprogram headers visible to clients, while hiding actual definitions
- Some primitive operations must be built into the language processor

**Design Issues:**

- Can abstract types be parameterized?
- What access controls are provided?

**3.10 Language Examples CO2, CO3, CO4**

**Language Examples: Ada**

- The encapsulation construct is called a package
- Specification package (the interface)
- Body package (implementation of the entities named in the specification)
- Information Hiding
- The spec package has two parts, public and private
- The name of the abstract type appears in the public part of the specification package.

This part may also include representations of unhidden types

- The representation of the abstract type appears in a part of the specification called the private part
- More restricted form with limited private types
- Private types have built-in operations for assignment and comparison
- Limited private types have NO built-in operations
- Reasons for the public/private spec package:
  1. The compiler must be able to see the representation after seeing only the spec package (it cannot see the private part)
  2. Clients must see the type name, but not the representation (they also cannot see the private part)
- Having part of the implementation details (the representation) in the spec package and part (the method bodies) in the body package is not good

**One solution: make all ADTs pointers**

Problems with this:

1. Difficulties with pointers
2. Object comparisons
3. Control of object allocation is lost

An Example in Ada

```
package Stack_Pack is
 type stack_type is limited
 private;max_size:
 constant := 100;
 function empty(stk: in stack_type) return Boolean;
 procedure push(stk: in out stack_type; elem:in
 Integer);procedure pop(stk: in out stack_type);
 function top(stk: in stack_type) return
 Integer;private -- hidden from clients
 type list_type is array (1..max_size)
 of Integer;type stack_type is record
 list: list_type;
```



```
topsub: Integer range
0..max_size) := 0;end record;
end Stack_Pack
```

3.

**Language Examples: C++**

- Based on C struct type and Simula 67 classes
- The class is the encapsulation device
- All of the class instances of a class share a single copy of the member functions



- Each instance of a class has its own copy of the class data members
- Instances can be static, stack dynamic, or heap dynamic
- Information Hiding
  - Private clause for hidden entities
  - Public clause for interface entities
  - Protected clause for inheritance (Chapter 12)
- Constructors:
  - Functions to initialize the data members of instances (they do not create the objects)
  - May also allocate storage if part of the object is heap-dynamic
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called

Name is the same as the class name
- Destructors
  - Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  - Implicitly called when the object's lifetime ends
  - Can be explicitly called
  - Name is the class name, preceded by a tilde (~) An Example in C++

```
class stack {
private:
int *stackPtr, maxlen, topPtr; public:
stack() { // a constructor stackPtr = new int [100]; maxlen = 99;
topPtr = -1;
};
~stack () {delete [] stackPtr;}; void push (int num) {...}; void pop () {...};
int top () {...};
int empty () {...};
}
```

#### Evaluation of ADTs in C++ and Ada

- C++ support for ADTs is similar to expressive power of Ada
- Both provide effective mechanisms for encapsulation and information hiding
- Ada packages are more general encapsulations; classes are types
- Friend functions or classes - to provide access to private members to some unrelated units or functions
  - Necessary in C++

#### Language Examples:

##### Java

- Similar to C++, except:
  - All user-defined types are classes
  - All objects are allocated from the heap and accessed through reference variables
  - Individual entities in classes have access control modifiers (private or public), rather than clauses
  - Java has a second scoping mechanism, package scope, which can be used in place of friends
- All entities in all classes in a package that do not have access control modifiers are visible throughout the package

An Example in Java

```
class StackClass { private:
private int [] *stackRef;
private int [] maxlen, topIndex; public StackClass() { // a constructor
stackRef = new int [100]; maxlen = 99;
topPtr = -1;
};
```

```
public void push (int num) {...}; public void pop () {...};
public int top () {...};
public boolean empty () {...};
}
```

### Language Examples: C#

- Based on C++ and Java
- Adds two access modifiers, internal and protected internal
- All class instances are heap dynamic
- Default constructors are available for all classes
- Garbage collection is used for most heap objects, so destructors are rarely used
- structs are lightweight classes that do not support inheritance
- Common solution to need for access to data members: accessor methods (getter and setter)
- C# provides properties as a way of implementing getters and setters without requiring explicit method calls

### C# Property Example

```
public class Weather {
 public int DegreeDays { /** DegreeDays is
 a propertyget {return degreeDays;}
 set {
 if(value < 0 || value > 30)
 Console.WriteLine("Value is out of range:
 {0}", value);else degreeDays = value;}
 }
 private int degreeDays;
 ...
}
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

### 3.10 Parameterized Abstract Data Types - CO4

- Parameterized ADTs allow designing an ADT that can store any type elements (among other things)
- Also known as generic classes
- C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs

#### Parameterized ADTs in Ada

- Ada Generic Packages
    - Make the stack type more flexible by making the element type and the size of the stack generic
- ```
generic
Max_Size: Positive;
type Elem_Type is private; package Generic_Stack is
Type Stack_Type is limited private;
function Top(Stk: in out StackType) return Elem_type;
...
end Generic_Stack;
Package Integer_Stack is new Generic_Stack(100,Integer); Package Float_Stack is new
Generic_Stack(100,Float);
```

Parameterized ADTs in C++

- Classes can be somewhat generic by writing parameterized constructor functions

```
class stack {
...
stack (int size) { stk_ptr = new int [size]; max_len = size - 1;
top = -1;
};
...
}
stack stk(100);
```

The stack element type can be parameterized by making the class a templated

```
class template <class Type> class stack {
private:
    Type
    *stack
    Ptr;
    const
    int
    maxLe
    n;int
    topPtr;
public:
    stack() {
        stackPtr = new
        Type[100];maxLen =
        99;
        topPtr = -1;
    }
    ...
}
```

Parameterized Classes in Java 5.0

- Generic parameters must be classes
- Most common generic types are the collection types, such as LinkedList and ArrayList
- Eliminate the need to cast objects that are removed
- Eliminate the problem of having multiple types in a structure

Parameterized Classes in C# 2005

- Similar to those of Java 5.0
- Elements of parameterized structures can be accessed through indexing

Summary of ADT

- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding
- Ada provides packages that simulate ADTs
- C++ data abstraction is provided by classes
- Java's data abstraction is similar to C++
- Ada, C++, Java 5.0, and C# 2005 support parameterized ADTs

3.12 ENCAPSULATION CONSTRUCTS

INTRODUCTION TO

ENCAPSULATION

Encapsulation Encapsulation is the process of combining or packaging data with functions and

PROGRAMMING LANGUAGES (CS3112PE)

thereby The concept of abstraction brings forth another related term known as encapsulation.

Original motivation:

☒ Large programs have two special needs:

Some means of organization, other than simply division into subprograms.

Some means of partial compilation (compilation units that are smaller than the whole program).

☒ Obvious solution:

a grouping of subprograms that are logically related into a unit that can be separately compiled.

These are called encapsulations.

Examples of Encapsulation Mechanisms

☒ Nested subprograms in some ALGOL-like languages (e.g., Pascal).



- ☒ FORTRAN 77 and C - Files containing one or more subprograms can be independently compiled.
- ☒ FORTRAN 90, C++, Ada (and other contemporary languages) - separately compilable modules.

3.13 Naming Encapsulation

- Naming encapsulations define name scopes that assist in avoiding these name conflicts.
- Naming encapsulations are logical encapsulations, in the sense that they need not be contiguous.
- Several different collections of code can be placed in the same namespace, even though they are stored in different places.

3.14 General Semantics of Calls and Returns

The subprogram call and return operations of a language are together called its subprogram linkage

A subprogram call has numerous actions associated with it.

Parameter passing methods Static local variables Execution status of calling program Transfer of control Subprogram nesting.

The General Semantics of Calls and Returns

- The **subprogram call and return** operations of a language are together called its *subprogram linkage*
- A subprogram call has numerous actions associated with it
 - Parameter passing methods
 - Static local variables
 - Execution status of calling program
 - Transfer of control
 - Subprogram nesting

Copyright © 2005 Addison-Wesley. All rights reserved.

2

3.15 Implementing “Simple” Subprograms:

Implementing “Simple” Subprograms: Call Semantics

- Save the execution status of the caller
- Carry out the parameter-passing process
- Pass the return address to the callee
- Transfer control to the callee

Implementing “Simple” Subprograms: Call Semantics

- Save the execution status of the caller
- Carry out the parameter-passing process
- Pass the return address to the callee
- Transfer control to the callee

Call Semantics

Save the execution status of the caller Carry out the parameter-passing process Pass the return address to the callee Transfer control to the callee

3.16 Implementing “Simple” Subprograms: Return Semantics

- If pass-by-value-result parameters are used, move the current values of those parameters to their corresponding actual parameters
- If it is a function, move the functional value to a place the caller can get it
- Restore the execution status of the caller
- Transfer control back to the caller

Implementing “Simple” Subprograms: Return Semantics

- If pass-by-value-result parameters are used, move the current values of those parameters to their corresponding actual parameters
- If it is a function, move the functional value to a place the caller can get it
- Restore the execution status of the caller
- Transfer control back to the caller

3.17 Implementing “Simple” Subprograms: Parts

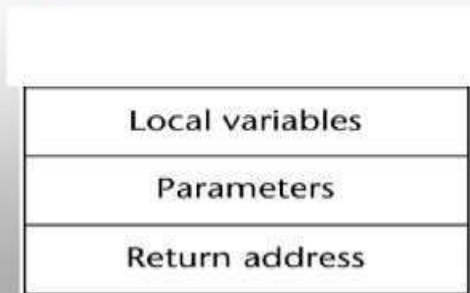
- Two separate parts: the actual code and the non-code part (local variables and data that can change)
- The format, or layout, of the non-code part of an executing subprogram is called an activation record
- An activation record instance is a concrete example of an activation record (the collection of data for a particular subprogram activation)

Implementing "Simple" Subprograms: Parts

- Two separate parts: the actual **code** and the **noncode** part (local variables and data that can change)
- The format, or layout, of the **noncode** part of an executing subprogram is called an **activation record**
- An **activation record instance** is a concrete example of an activation record (the collection of data for a particular subprogram activation)

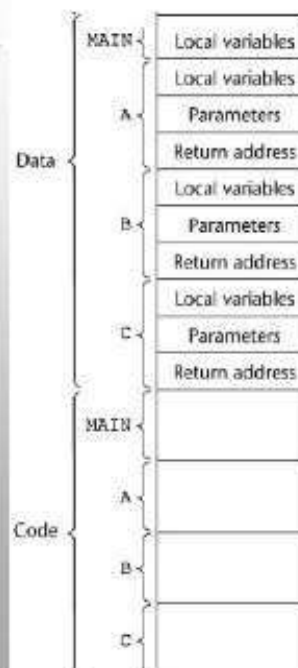
An Activation Record for "Simple" Subprograms

An Activation Record for "Simple" Subprograms



Code and Activation Records of a Program with "Simple" Subprograms

Code and Activation Records of a Program with "Simple" Subprograms



Copyright © 2000 Addison-Wesley. All rights reserved.

7

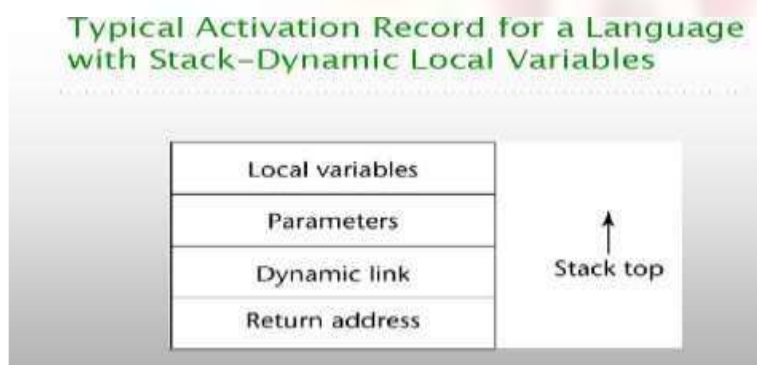
3.18 Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
 - The compiler must generate code to cause implicit allocation and de-allocation of local variables
 - Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
 - The compiler must generate code to cause implicit allocation and de-allocation of local variables
 - Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

Typical Activation Record for a Language with Stack-Dynamic Local Variables



3.19 Implementing Subprograms with Stack.

Dynamic Local Variables:

Activation Record

- The activation record format is static, but its size may be dynamic
- The dynamic link points to the top of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called
- Run-time stack

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record **format is static**, but its **size may be dynamic**
- The *dynamic link* points to the top of an instance of **the activation record of the caller**
- An activation record instance is dynamically created when a subprogram is called
- Run-time stack

An Example: C Function

```
void sub(float total, int part)
{
    int list[4];
    float sum;
    ...
}
```

Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Return address	
Return address	



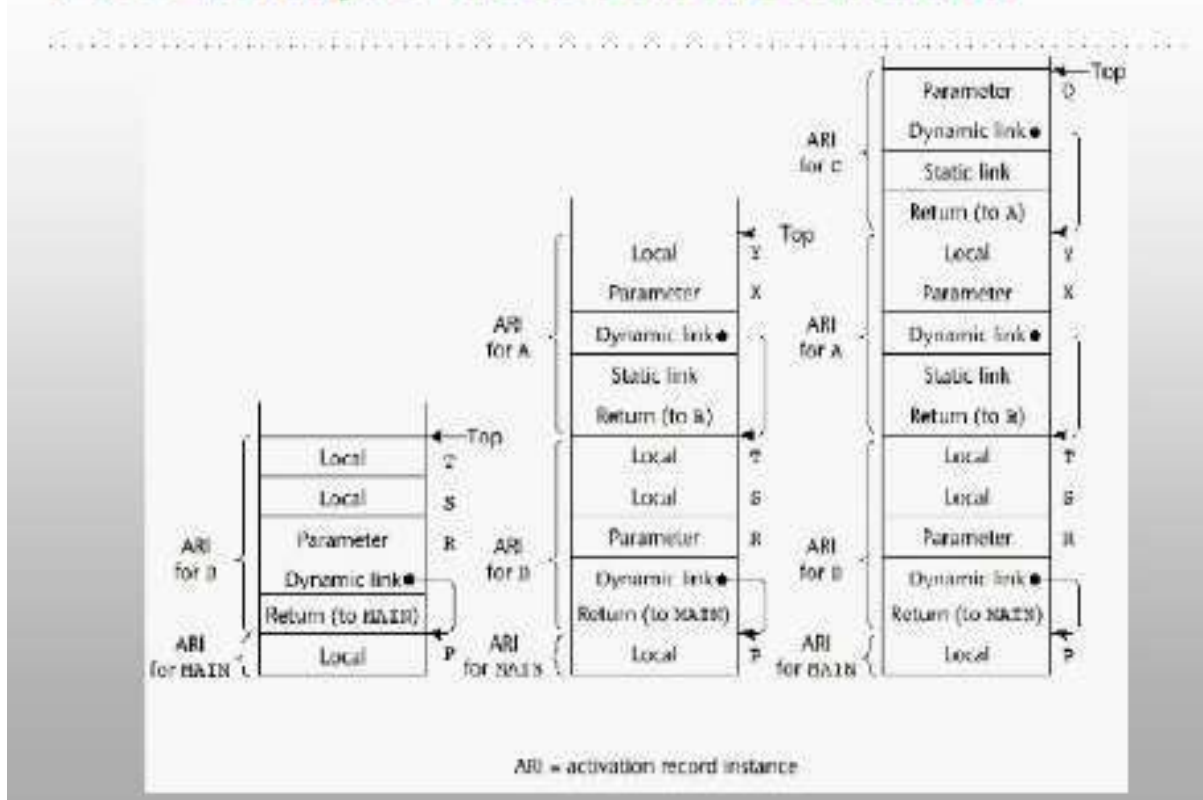
An Example Without Recursion

```
void A(int x) {  
    int y;  
    ...  
    C(y);  
    ...  
}  
void B(float r) {  
    int s, t;  
    ...  
    A(s);  
    ...  
}  
void C(int q) {  
    ...  
}  
void main() {  
    float p;  
    ...  
    B(p);  
    ...  
}
```

main calls B
B calls A
A calls C

The logo for NIRGM (National Institute of Research in Grammatical Morphology) is displayed. It features the acronym 'NIRGM' in a large, bold, serif font. The letters are filled with a light purple color and have a subtle gradient. A small circular emblem is positioned in the center of the letter 'R'. The logo is set against a white background with a faint, light purple border.

An Example Without Recursion



Dynamic Chain and Local Offset

- The collection of dynamic links in the stack at a given time is called the dynamic chain, or call chain
- Local variables can be accessed by their offset from the beginning of the activation record. This offset is called the local_offset
- The local_offset of a local variable can be determined by the compiler at compile time



An Example With Recursion

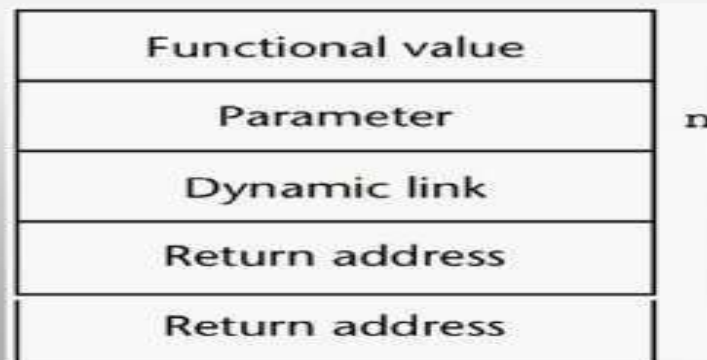
- The activation record used in the previous example supports recursion, e.g.

```

int factorial (int n) {
    <-----1
    if (n <= 1) return 1;
    else return (n * factorial(n - 1));
    <-----2
}
void main() {
    int value;
    value = factorial(3);
    <-----3
}

```

Activation Record for factorial



Nested Subprograms

- Some non-C-based static-scoped languages (e. g. , Fortran 95, Ada, Java. Script) use stack-dynamic local variables and allow subprograms to be nested
- All variables that can be non-locally accessed reside in some activation record instance in the stack
- The process of locating a non-local reference:
 - Find the correct activation record instance
 - Determine the correct offset within that activation record instance

Locating a Non-local Reference

- Finding the offset is easy
- Finding the correct activation record instance
 - Static semantic rules guarantee that all nonlocal variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made

Static Scoping

- A static chain is a chain of static links that connects certain activation record instances

- The static link in an activation record instance for subprogram A points to one of the activation record instances of A's static parent
- The static chain from an activation record instance connects it to all of its static ancestors

• **Stack Contents at Position 1** $\bar{7}$
 Dynamic Links; MAIN_2 calls

BIGSUB calls

SUB 2 calls

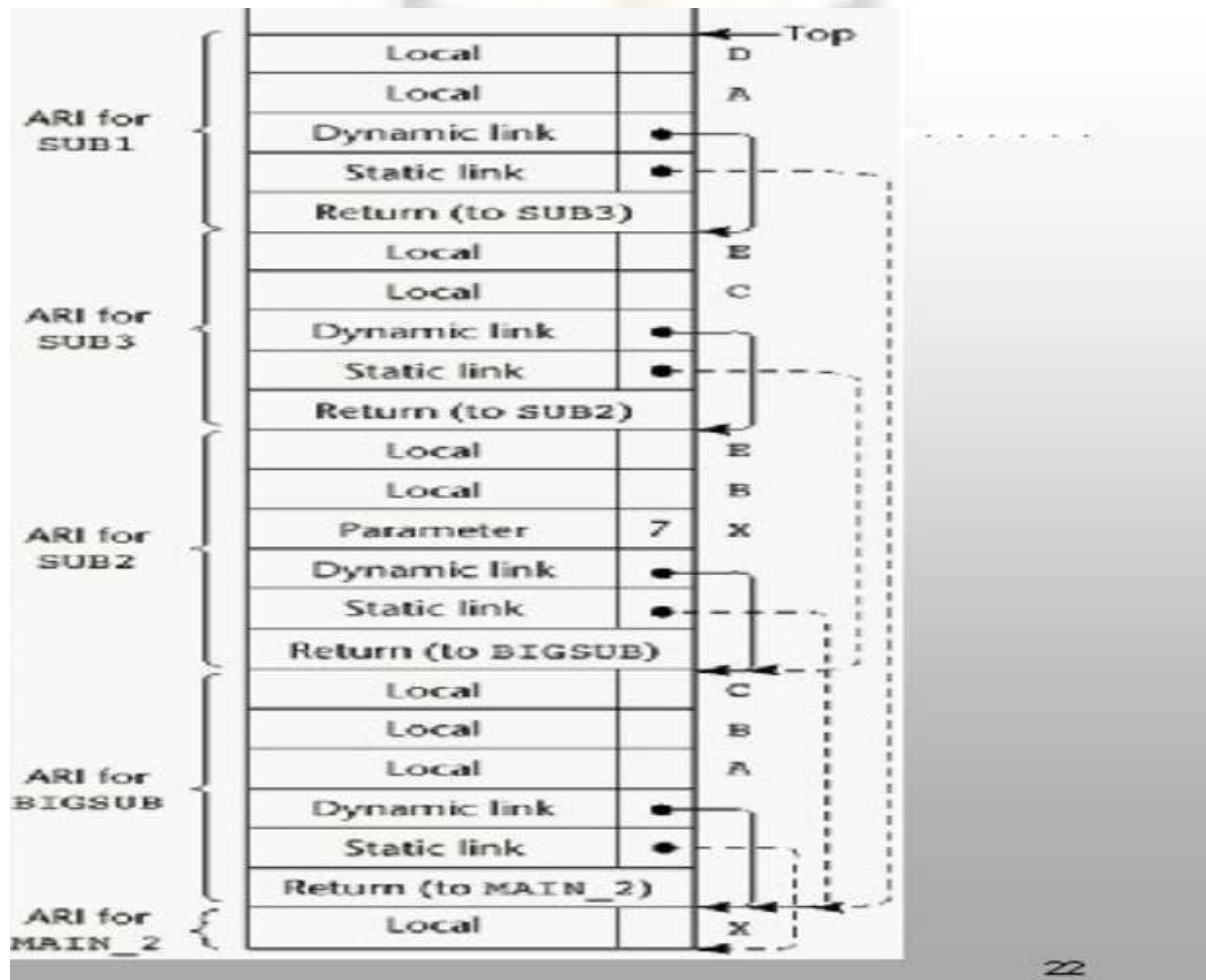
SUB 3 calls SUB 1

$\bar{7}$ Static Links:

SUB 1 static parent:

BIGSUB 2 static parent:

BIGSUB 3 static parent: SUB 2



Displays

- An alternative to static chains
- Static links are stored in a single array called a display
- The contents of the display at any given time is a list of addresses of the accessible activation record instances

Blocks

- Blocks are user-specified local scopes for variables

```
{int temp;
  temp = list [upper];
  list [upper] = list [lower];
  list [lower] = temp
}
```

- The lifetime of temp in the above example begins when control enters the block
- An advantage of using a local variable like temp is that it cannot interfere with any other variable with the same name

Implementing Blocks

Two Methods:

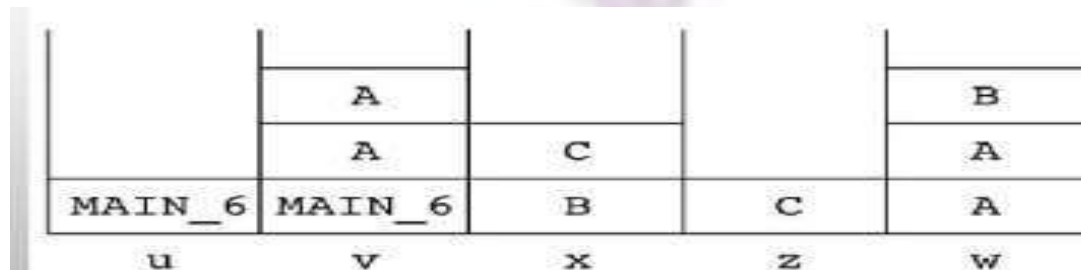
1. Treat blocks as parameter-less subprograms that are always called from the same location – Every block has an activation record; an instance is created every time the block is executed
2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Dynamic Scoping

- Deep Access: non-local references are found by searching the activation record instances on the dynamic chain
- Shallow Access: put locals in a central place
 - One stack for each variable name
 - Central table with an entry for each variable name

Using Shallow Access to Implement Dynamic

Scoping



(The names in the stack cells indicate the program units of the variable declaration.)

SUMMARY

- Subprogram linkage semantics requires many action by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex
- Subprograms with stack-dynamic local variables and nested subprograms have two components – actual code – activation record
- Activation record instances contain formal parameters and local variables among other things
- Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or thru some central variable table method

UNIT - 4

4.1 Concurrency

- Concurrency can occur at four levels:
 - Machine instruction level
 - High-level language statement level
 - Unit level
 - Program level
- Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

Multiprocessor Architectures

- Late 1950s - one general-purpose processor and one or more special purpose processors for input and output operations
- Early 1960s - multiple complete processors, used for program-level concurrency
- Mid-1960s - multiple partial processors, used for instruction-level concurrency
- Single-Instruction Multiple-Data (SIMD) machines
- Multiple-Instruction Multiple-Data (MIMD) machines
 - Independent processors that can be synchronized (unit-level concurrency)

Categories of Concurrency

- A thread of control in a program is the sequence of program points reached as control flows through the program
- Categories of Concurrency:
 - Physical concurrency - Multiple independent processors (multiple threads of control)
 - Logical concurrency - The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)
- Coroutines (quasi-concurrency) have a single thread of control

Motivations for Studying Concurrency

- Involves a different way of designing software that can be very useful— many real-world situations involve concurrency
- Multiprocessor computers capable of physical concurrency are now widely used

Subprogram-Level Concurrency

- A task or process is a program unit that can be in concurrent execution with other program units
- Tasks differ from ordinary subprograms in that:
 - A task may be implicitly started
 - When a program unit starts the execution of a task, it is not necessarily suspended
 - When a task's execution is completed, control may not return to the caller
- Tasks usually work together

Two General Categories of Tasks

- Heavyweight tasks execute in their own address space
- Lightweight tasks all run in the same address space
- A task is disjoint if it does not communicate with or affect the execution of any other task in the program in any way

Task Synchronization

- A mechanism that controls the order in which tasks execute
- Two kinds of synchronization
 - Cooperation synchronization
 - Competition synchronization
- Task communication is necessary for synchronization, provided by:
 - Shared nonlocal variables

- Parameters
Message passing

Kinds of synchronization

- *Cooperation*: Task A must wait for task B to complete some specific activity before task A can continue its execution, e.g., the producer-consumer problem
- *Competition*: Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter
- Competition is usually provided by mutually exclusive access (approaches are discussed later)

Scheduler

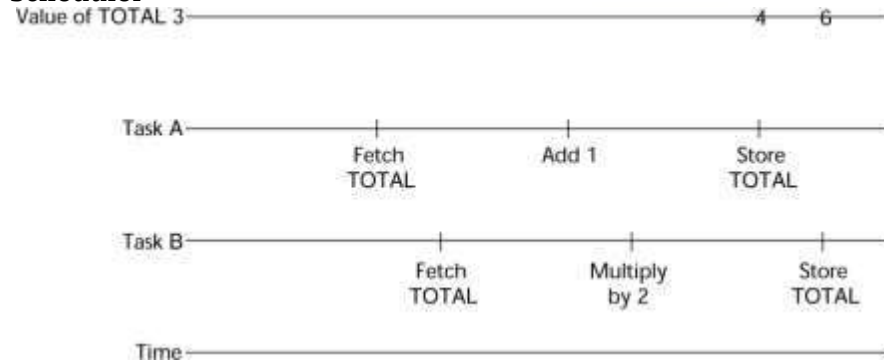


Figure 6.1 Need for Competition Synchronization

- Providing synchronization requires a mechanism for delaying task execution
- Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors

Task Execution States

- New - created but not yet started
- Ready - ready to run but not currently running (no available processor)
- Running
- Blocked - has been running, but cannot now continue (usually waiting for some event to occur)
- Dead - no longer active in any sense

Liveness and Deadlock

- Liveness is a characteristic that a program unit may or may not have
- In sequential code, it means the unit will eventually complete its execution
- In a concurrent environment, a task can easily lose its liveness
- If all tasks in a concurrent environment lose their liveness, it is called deadlock

Design Issues for Concurrency

- Competition and cooperation synchronization
- Controlling task scheduling
- How and when tasks start and end execution
- How and when are tasks created

Methods of Providing

Synchronization

- Semaphores
- Monitors
- Message Passing

Semaphores

- Dijkstra - 1965
- A semaphore is a data structure consisting of a counter and a queue for storing task descriptors

- Semaphores can be used to implement guards on the code that accesses shared data structures
- Semaphores have only two operations, wait and release (originally called P and V by Dijkstra)
- Semaphores can be used to provide both competition and cooperation synchronization

Cooperation Synchronization with Semaphores

- Example: A shared buffer
- The buffer is implemented as an ADT with the operations DEPOSIT and FETCH as the only ways to access the buffer
- Use two semaphores for cooperation: emptyspots and fullspots
- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer
- DEPOSIT must first check emptyspots to see if there is room in the buffer
- If there is room, the counter of emptyspots is decremented and the value is inserted
- If there is no room, the caller is stored in the queue of emptyspots
- When DEPOSIT is finished, it must increment the counter of fullspots
- FETCH must first check fullspots to see if there is a value
- If there is a full spot, the counter of fullspots is decremented and the value is removed
- If there are no values in the buffer, the caller must be placed in the queue of fullspots
- When FETCH is finished, it increments the counter of emptyspots
- The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named wait and release

Semaphores: Wait Operation

```
wait(aSemaphore)
    if aSemaphore's counter > 0
        then decrement
         aSemaphore's
          counter
    else
        put the caller in
         aSemaphore's queue attempt
         to transfer control to a ready
          task
        -- if the task ready queue is empty,
        -- deadlock occurs
    end
```

Semaphores: Release Operation

```
release(aSemaphore)
    if aSemaphore's queue is empty then increment aSemaphore's counter
    else
        put the calling task in the task ready queue transfer control
         to a task from aSemaphore's queue
    end
```

put the calling task in the task ready queue transfer control to a task from aSemaphore's queue

Producer Consumer Code

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
loop
    -- produce VALUE --
```



```
PROGRAMMING LANGUAGES (CS3112PE)  
wait (emptyspots); {wait for space}  
DEPOSIT(VALUE);  
release(fullspots); {increase filled}  
end loop;  
end producer;
```



Producer Consumer Code

```

    task consumer;
        loop
            wait (fullspots);{wait till
            not empty}}
            FETCH(VALUE);
            release(emptyspots); {increase empty}
        -- consume VALUE --
    end loop;
end consumer;

```

Competition Synchronization with Semaphores

- A third semaphore, named access, is used to control access (competition synchronization)
- The counter of access will only have the values 0 and 1
- Such a semaphore is called a binary semaphore
- Note that wait and release must be atomic!

Producer Consumer Code

```

semaphore access, fullspots, emptyspots;
access.count = 0;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
loop
    -- produce VALUE --
    wait(emptyspots); {wait for space}
    wait(access); {wait for access}
    DEPOSIT(VALUE);
    release(access); {relinquish access}
    release(fullspots); {increase filled}
end loop;
end producer;

```

Producer Consumer Code

```

    task consumer;
        loop
            wait(fullspots);{wait till not empty}
            wait(access); {wait for access}
            FETCH(VALUE);
            release(access); {relinquish access}
            release(emptyspots); {increase empty}
        -- consume VALUE --
    end loop;
end consumer;

```

Evaluation of Semaphores

- Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer will overflow if the wait of fullspots is left out
- Misuse of semaphores can cause failures in competition synchronization, e.g., the program will deadlock if the release of access is left out

Monitors

- Ada, Java, C#
- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data

Competition Synchronization

- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor
- Monitor implementation guarantee synchronized access by allowing only one access at a time

PROGRAMMING LANGUAGES (CS3112PE)

- Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

Cooperation Synchronization

- Cooperation between processes is still a programming task
- Programmer must guarantee that a shared buffer does not experience underflow or overflow



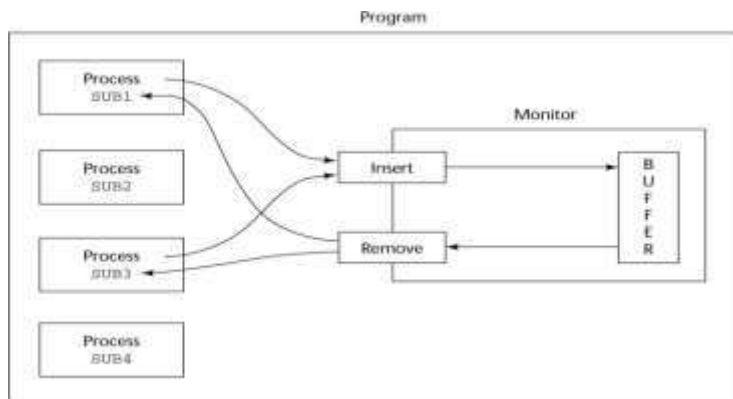


Figure 6.2 Cooperation Synchronization

Evaluation of Monitors

- A better way to provide competition synchronization than are semaphores
- Semaphores can be used to implement monitors
- Monitors can be used to implement semaphores
- Support for cooperation synchronization is very similar as with semaphores, so it has the same problems

Message Passing

- Message passing is a general model for concurrency
 - It can model both semaphores and monitors
 - It is not just for competition synchronization
- Central idea: task communication is like seeing a doctor--most of the time she waits for you or you wait for her, but when you are both ready, you get together, or rendezvous

Message Passing Rendezvous

- To support concurrent tasks with message passing, a language needs:
 - A mechanism to allow a task to indicate when it is willing to accept messages
 - A way to remember who is waiting to have its message accepted and some “fair” way of choosing the next message
- When a sender task’s message is accepted by a receiver task, the actual message transmission is called a rendezvous

Ada Support for Concurrency

- The Ada 83 Message-Passing Model
 - Ada tasks have specification and body parts, like packages; the spec has the interface, which is the collection of entry points:
 - **task Task_Example is**
 - **entry ENTRY_1 (Item : in Integer);end**
 - Task_Example;**
 - Task Body**
 - The body task describes the action that takes place when a rendezvous occurs
 - A task that sends a message is suspended while waiting for the message to be accepted and during the rendezvous
 - Entry points in the spec are described with accept clauses in the body accept entry_name (formal parameters) do
 - ...
 - end entry_name

Example of a Task Body

```
task body Task_Example is begin
loop
accept Entry_1 (Item: in Float) do
...
```

```
end Entry_1;
end loop; end Task_Example;
```

Ada Message Passing Semantics

- The task executes to the top of the accept clause and waits for a message
- During execution of the accept clause, the sender is suspended
- accept parameters can transmit information in either or both directions
- Every accept clause has an associated queue to store waiting messages

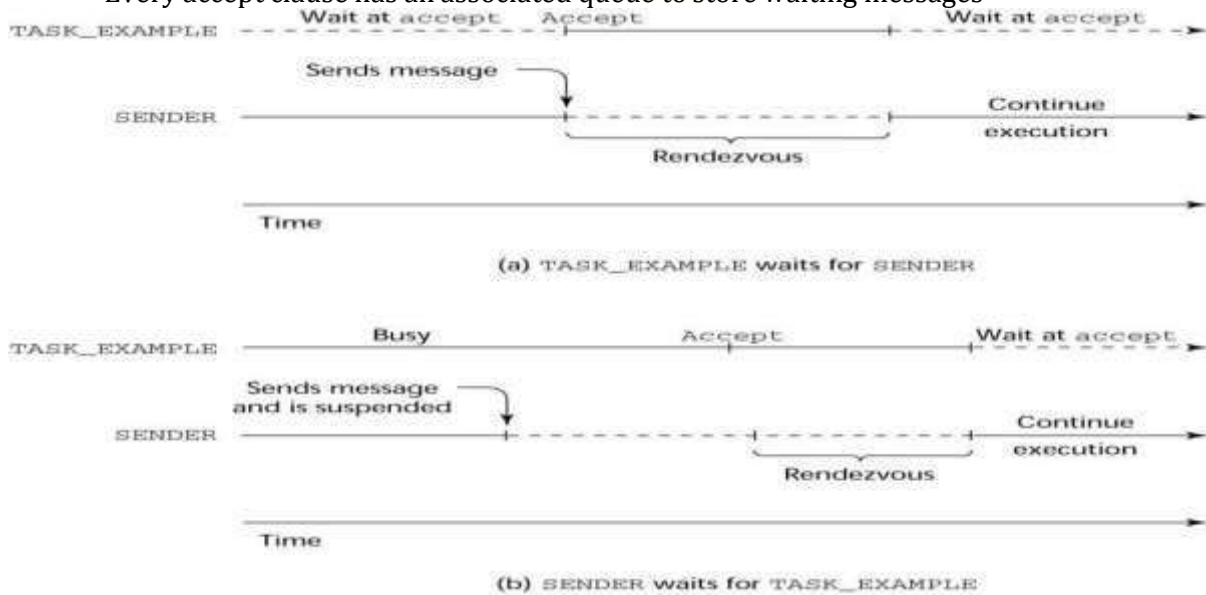


Figure 6.3 Rendezvous Time Lines

Message Passing: Server/Actor Tasks

- A task that has accept clauses, but no other code is called a server task (the example above is a server task)
- A task without accept clauses is called an actor task
- An actor task can send messages to other tasks
- Note: A sender must know the entry name of the receiver, but not vice versa (asymmetric)

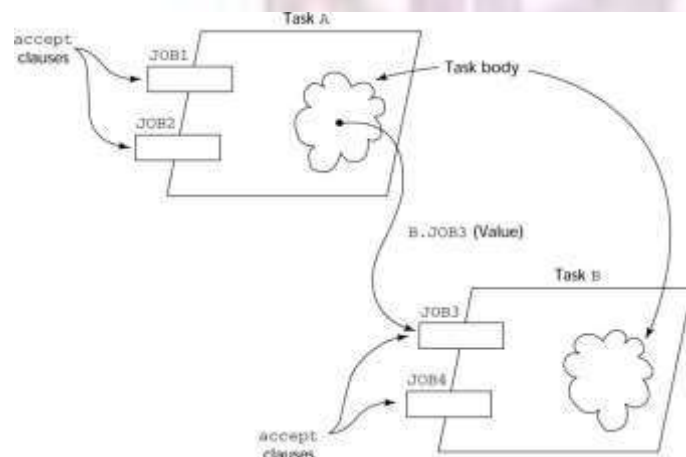


Figure 6.4 Graphical Representation of a Rendezvous

Example: Actor Task

```
task Water_Monitor; -- specification
task body body Water_Monitor is -- body
begin
loop
if Water_Level > Max_Level then Sound_Alarm;
```

end if;

end loop; end Water_Monitor;

Multiple Entry Points

delay 1.0; -- No further execution
-- for at least 1 second

- Tasks can have more than one entry point
- The specification task has an entry clause for each
- The task body has an accept clause for each entry clause, placed in a select clause, which is in a loop

A Task with Multiple Entries

```
task body Teller is loop
select
accept Drive_Up(formal params) do
...
end Drive_Up;
...
or
accept Walk_Up(formal params) do
...
end Walk_Up;
...
end select; end loop;
end Teller;
```

Semantics of Tasks with Multiple accept Clauses

- If exactly one entry queue is nonempty, choose a message from it
- If more than one entry queue is nonempty, choose one, nondeterministically, from which to accept a message
- If all are empty, wait
- The construct is often called a selective wait
- Extended accept clause - code following the clause, but before the next clause
- Executed concurrently with the caller

Cooperation Synchronization with Message Passing

- Provided by Guarded accept clauses
- ```
when not Full(Buffer) => accept Deposit (New_Value) do
```
- An accept clause with a with a when clause is either open or closed
  - A clause whose guard is true is called open
  - A clause whose guard is false is called closed
  - A clause without a guard is always open

### Semantics of select with Guarded accept Clauses:

- select first checks the guards on all clauses
- If exactly one is open, its queue is checked for messages
- If more than one are open, non-deterministically choose a queue among them to check for messages
- If all are closed, it is a runtime error
- A select clause can include an else clause to avoid the error



- When the else clause completes, the loop repeats

#### Example of a Task with Guarded accept Clauses

• Note: The station may be out of gas and there may or may not be a position available in the garage  
 task Gas\_Station\_Attendant is  
 entry Service\_Island (Car : Car\_Type); entry Garage (Car : Car\_Type); end  
 Gas\_Station\_Attendant;

#### Example of a Task with Guarded accept Clauses

```
task body Gas_Station_Attendant is
begin
 loop
 select
 when Gas_Available =>
 accept Service_Island (Car : Car_Type) do
 Fill_With_Gas (Car);
 end Service_Island;
 or
 when Garage_Available =>
 accept Garage (Car : Car_Type) do
 Fix (Car);
 end Garage;
 else
 Sleep;
 end select;
 end loop;
end Gas_S_A;
```

**Competition Synchronization with Message Passing**

- Modeling mutually exclusive access to shared data
- Example--a shared buffer
- Encapsulate the buffer and its operations in a task
- Competition synchronization is implicit in the semantics of accept clauses
- Only one accept clause in a task can be active at any given time

**Task Termination**

- The execution of a task is completed if control has reached the end of its code body
- If a task has created no dependent tasks and is completed, it is terminated
- If a task has created dependent tasks and is completed, it is not terminated until all its dependent tasks are terminated

**The terminate Clause**

- A terminate clause in a select is just a terminate statement
- A terminate clause is selected when no accept clause is open
- When a terminate is selected in a task, the task is terminated only when its master and all of the dependents of its master are either completed or are waiting at a terminate
- A block or subprogram is not left until all of its dependent tasks are terminated

**Message Passing Priorities**

- The priority of any task can be set with the pragma priority pragma Priority (expression);
- The priority of a task applies to it only when it is in the task ready queue

**Binary Semaphores**



- For situations where the data to which access is to be controlled is NOT encapsulated in a task  
task Binary\_Semaphore is entry Wait;  
entry release; end Binary\_Semaphore;  
task body Binary\_Semaphore is begin  
loop  
accept Wait; accept Release; end loop;  
end Binary\_Semaphore;

### Concurrency in Ada 95

- Ada 95 includes Ada 83 features for concurrency, plus two new features
- Protected objects: A more efficient way of implementing shared data to allow access to a shared data structure to be done without rendezvous
- Asynchronous communication

### Ada 95: Protected Objects

- A protected object is similar to an abstract data type
- Access to a protected object is either through messages passed to entries, as with a task, or through protected subprograms
- A protected procedure provides mutually exclusive read-write access to protected objects
- A protected function provides concurrent read-only access to protected objects

### Asynchronous Communication

- Provided through asynchronous select structures
- An asynchronous select has two triggering alternatives, an entry clause or a delay
- The entry clause is triggered when sent a message
- The delay clause is triggered when its time limit is reached

### Evaluation of the Ada

- Message passing model of concurrency is powerful and general
- Protected objects are a better way to provide synchronized shared data
- In the absence of distributed processors, the choice between monitors and tasks with message passing is somewhat a matter of taste
- For distributed systems, message passing is a better model for concurrency

### Java Threads

- The concurrent units in Java are methods named run
- A run method code can be in concurrent execution with other such methods
- The process in which the run methods execute is called a thread

```
Class myThread extends Thread{
public void run () {...}
}
```

...

```
Thread myTh = new MyThread (); myTh.start();
```

### Controlling Thread Execution

- The Thread class has several methods to control the execution of threads
- The yield is a request from the running thread to voluntarily surrender the processor
- The sleep method can be used by the caller of the method to block the thread
- The join method is used to force a method to delay its execution until the run method of another thread has completed its execution

### Thread Priorities

- A thread's default priority is the same as the thread that create it

- If main creates a thread, its default priority is NORM\_PRIORITY
- Threads defined two other priority constants, MAX\_PRIORITY and MIN\_PRIORITY
- The priority of a thread can be changed with the methods setPriority

### Competition Synchronization with Java Threads

- A method that includes the synchronized modifier disallows any other method from running on the object while it is in execution

...

```
public synchronized void deposit(int i) {...} public synchronized int fetch() {...}
```

...

- The above two methods are synchronized which prevents them from interfering with each other

- If only a part of a method must be run without interference, it can be synchronized through synchronized statement

```
synchronized (expression)
statement
```

### Cooperation Synchronization with Java Threads

- Cooperation synchronization in Java is achieved via wait, notify, and notifyAll methods
- All methods are defined in Object, which is the root class in Java, so all objects inherit them
- The wait method must be called in a loop
- The notify method is called to tell one waiting thread that the event it was waiting has happened
- The notifyAll method awakens all of the threads on the object's wait list

### Java's Thread Evaluation

- Java's support for concurrency is relatively simple but effective
- Not as powerful as Ada's tasks

### C# Threads

- Loosely based on Java but there are significant differences
- Basic thread operations
- Any method can run in its own thread
- A thread is created by creating a Thread object
- Creating a thread does not start its concurrent execution; it must be requested through the Start method
- A thread can be made to wait for another thread to finish with Join
- A thread can be suspended with Sleep
- A thread can be terminated with Abort

### Synchronizing Threads

- Three ways to synchronize C# threads
- The Interlocked class
- Used when the only operations that need to be synchronized are incrementing or decrementing of an integer
- The lock statement
- Used to mark a critical section of code in a thread lock (expression) {...}
- The Monitor class
- Provides four methods that can be used to provide more sophisticated synchronization

### C#'s Concurrency Evaluation

- An advance over Java threads, e.g., any method can run its own thread

- Thread termination is cleaner than in Java
- Synchronization is more sophisticated

### **Statement-Level Concurrency**

- Objective: Provide a mechanism that the programmer can use to inform compiler of ways it can map the program onto multiprocessor architecture
- Minimize communication among processors and the memories of the other processors

### **High-Performance Fortran**

- A collection of extensions that allow the programmer to provide information to the compiler to help it optimize code for multiprocessor computers
- Specify the number of processors, the distribution of data over the memories of those processors, and the alignment of data

### **Primary HPF Specifications**

- Number of processors  
!HPF\$ PROCESSORS procs (n)
- Distribution of data  
!HPF\$ DISTRIBUTE (kind) ONTO procs :: identifier\_list  
– kind can be BLOCK (distribute data to processors in blocks) or CYCLIC (distribute data to processors one element at a time)
- Relate the distribution of one array with that of another  
ALIGN array1\_element WITH array2\_element

### **Concurrency in Functional Languages**

Concurrency is naturally divided into:

- 1) Instruction level concurrency
- 2) Statement level concurrency (executing two or more statements simultaneously)
- 3) Program unit level concurrency (execute two or more subprogram units simultaneously) and
- 4) Program level concurrency (executing two or more programs simultaneously).

### **Categories of Concurrency:**

**Physical concurrency** - Multiple independent processors ( multiple threads of control)

**2. Logical concurrency** - The appearance of physical concurrency is presented by time sharing one processor (software can be designed as if there were multiple threads of control)

- Coroutines provide only quasicurrency

### **Reasons to Study Concurrency:**

It involves a new way of designing software that can be very useful--many real-world situation involve concurrency

2. Computers capable of physical concurrency are now widely used

### **Fundamentals (for stmt-level concurrency) :**

Def: A task is a program unit that can be in concurrent execution with other program units

Tasks differ from ordinary subprograms in that:

A task may be implicitly started

When a program unit starts the execution of a task, it is not necessarily suspended

When a task's execution is completed, control may not return to the caller

**Def:** A task is disjoint if it does not communicate with or affect the execution of any other task in the program in any way Task communication is necessary for synchronization

### Statement-Level Concurrency Example

```
REAL list_1(1000), list_2(1000)
INTEGER list_3(500), list_4(501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs ::
list_1, list_2
!HPF$ ALIGN list_1(index) WITH
list_4 (index+1)
...
list_1 (index) = list_2(index)
list_3(index) = list_4(index+1)
```

FORALL statement is used to specify a list of statements that may be executed concurrently

FORALL (index = 1:1000)

list\_1(index) = list\_2(index)

- Specifies that all 1,000 RHSs of the assignments can be evaluated before any assignment takes place

### Summary

- Concurrent execution can be at the instruction, statement, or subprogram level
- Physical concurrency: when multiple processors are used to execute concurrent units
- Logical concurrency: concurrent units are executed on a single processor
- Two primary facilities to support subprogram concurrency: competition synchronization and cooperation synchronization
- Mechanisms: semaphores, monitors, rendezvous, threads
- High-Performance Fortran provides statements for specifying how data is to be distributed over the memory units connected to multiple processors

## Exception Handling & Logic Programming Language

### Introduction to Exception Handling & Event Handling

- In a language without exception handling
  - When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated
- In a language with exception handling
  - Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing

### 4.7 Basic Concepts - CO3

- Many languages allow programs to trap input/output errors (including EOF)
- An exception is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
- The special processing that may be required after detection of an exception is called exception handling
- The exception handling code unit is called an exception handler

### Exception Handling Alternatives

- An exception is raised when its associated event occurs
- A language that does not have exception handling capabilities can still define, detect,



raise, and handle exceptions (user defined, software detected)

- Alternatives:
  - Send an auxiliary parameter or use the return value to indicate the return status of a subprogram
  - Pass an exception handling subprogram to all subprograms

**Advantages of Built-in Exception Handling**

- Error detection code is tedious to write and it clutters the program
- Exception handling encourages programmers to consider many different possible errors
- Exception propagation allows a high level of reuse of exception handling code

**Design Issues**

- How are user-defined exceptions specified?
- Should there be default exception handlers for programs that do not provide their own?
- Can built-in exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that can be handled?
- Are there any built-in exceptions?
- How can exceptions be disabled, if at all?
- How and where exception handlers specified and what are their scope?
- How is an exception occurrence bound to an exception handler?
- Can information about the exception be passed to the handler?
- Where does execution continue, if at all, after an exception handler completes its execution? (continuation vs. resumption)
- Is some form of finalization provided?

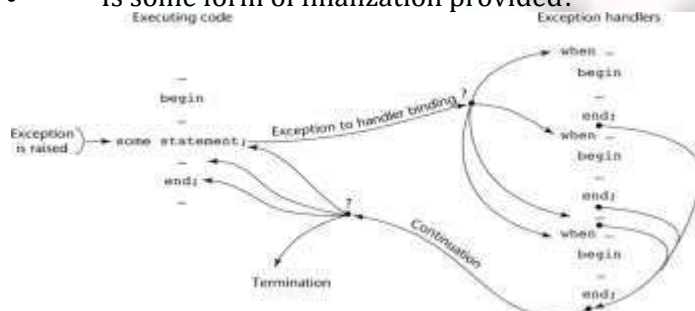


Figure 7.1 Exception Handling Control Flow

**4.8 Exception Handling in Ada – C03**

- The frame of an exception handler in Ada is either a subprogram body, a package body, a task, or a block
- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters

**Ada Exception Handlers**

• Handler form:  
 when exception\_choice{|exception\_choice} => statement\_sequence  
 ...  
 [when others => statement\_sequence]

exception\_choice form: exception\_name | others

- Handlers are placed at the end of the block or unit in which they occur

**Binding Exceptions to Handlers**

- If the block or unit in which an exception is raised does not have a handler for that exception, the exception is propagated elsewhere to be handled
  - Procedures - propagate it to the caller

- Blocks - propagate it to the scope in which it appears
- Package body - propagate it to the declaration part of the unit that declared the package (if it is a library unit, the program is terminated)
- Task - no propagation; if it has a handler, execute it; in either case, mark it "completed"

**Continuation**

- The block or unit that raises an exception but does not handle it is always terminated (also any block or unit to which it is propagated that does not handle it)

**Other Design Choices**

- User-defined Exceptions form:  
exception\_name\_list : exception;
- Raising Exceptions form:  
raise [exception\_name]
- (the exception name is not required if it is in a handler--in this case, it propagates the same exception)
- Exception conditions can be disabled with:  
pragma SUPPRESS(exception\_list)

**Predefined Exceptions**

- CONSTRAINT\_ERROR - index constraints, range constraints, etc.
- NUMERIC\_ERROR - numeric operation cannot return a correct value (overflow, division by zero, etc.)
- PROGRAM\_ERROR - call to a subprogram whose body has not been elaborated
- STORAGE\_ERROR - system runs out of heap
- TASKING\_ERROR - an error associated with tasks

**Evaluation**

- The Ada design for exception handling embodies the state-of-the-art in language design in 1980
- A significant advance over PL/I
- Ada was the only widely used language with exception handling until it was added to C++

**4.9 Exception Handling in C++ - CO3**

- Added to C++ in 1990
- Design is based on that of CLU, Ada, and ML

**C++ Exception Handlers**

- Exception Handlers Form:  
try {  
-- code that is expected to raise an exception  
}  
catch (formal parameter) {  
-- handler code  
}  
...  
catch (formal parameter) {  
-- handler code  
}

**The catch Function**

- catch is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique
- The formal parameter need not have a variable
- It can be simply a type name to distinguish the handler it is in from others
- The formal parameter can be used to transfer information to the handler
- The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

### Throwing Exceptions

- Exceptions are all raised explicitly by the statement: throw [expression];
- The brackets are metasymbols
- A throw without an operand can only appear in a handler; when it appears, it simply re-raises the exception, which is then handled elsewhere
- The type of the expression disambiguates the intended handler

### Unhandled Exceptions

- An unhandled exception is propagated to the caller of the function in which it is raised
- This propagation continues to the main function

### Continuation

- After a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers of which it is an element
- Other design choices
- All exceptions are user-defined
- Exceptions are neither specified nor declared
- Functions can list the exceptions they may raise
- Without a specification, a function can raise any exception (the throw clause)

### Evaluation

- It is odd that exceptions are not named and that hardware- and system software-detectable exceptions cannot be handled
- Binding exceptions to handlers through the type of the parameter certainly does not promote readability

### 4.13 Exception Handling in Java - CO3

- Based on that of C++, but more in line with OOP philosophy
- All exceptions are objects of classes that are descendants of the Throwable class

#### Classes of Exceptions

- The Java library includes two subclasses of Throwable :
    - Error
      - o Thrown by the Java interpreter for events such as heap overflow
      - o Never handled by user programs
    - Exception
      - o User-defined exceptions are usually subclasses of this
      - o Has two predefined subclasses, IOException and RuntimeException
- e.g., ArrayIndexOutOfBoundsException and NullPointerException

#### Java Exception Handlers

- Like those of C++, except every catch requires a named parameter and all parameters must be descendants of Throwable
- Syntax of try clause is exactly that of C++
- Exceptions are thrown with throw, as in C++, but often the throw includes the new operator to create the object, as in: throw new MyException();

#### Binding Exceptions to Handlers

- Binding an exception to a handler is simpler in Java than it is in C++
  - An exception is bound to the first handler with a parameter is the same class as the thrown object or an ancestor of it
- An exception can be handled and rethrown by including a throw in the handler (a handler could also throw a different exception)

#### Continuation

- If no handler is found in the method, the exception is propagated to the method's caller
- If no handler is found (all the way to main), the program is terminated

- To ensure that all exceptions are caught, a handler can be included in any try construct that catches all exceptions
- Simply use an Exception class parameter
- Of course, it must be the last in the try construct

### Checked and Unchecked Exceptions

- The Java throws clause is quite different from the throw clause of C++
- Exceptions of class Error and RuntimeException and all of their descendants are called unchecked exceptions; all other exceptions are called checked exceptions
- Checked exceptions that may be thrown by a method must be either:
  - Listed in the throws clause, or
  - Handled in the method

### Other Design Choices

- A method cannot declare more exceptions in its throws clause than the method it overrides
- A method that calls a method that lists a particular checked exception in its throws clause has three alternatives for dealing with that exception:
  - Catch and handle the exception
  - Catch the exception and throw an exception that is listed in its own throws clause
  - Declare it in its throws clause and do not handle it

### The finally Clause

- Can appear at the end of a try construct
- Form:
 

```
finally {..}
```
- Purpose: To specify code that is to be executed, regardless of what happens in the try construct

### Example

- A try construct with a finally clause can be used outside exception handling
- ```
try {
for (index = 0; index < 100; index++) {
...
if (...) {
return;
} /** end of if
} /** end of try clause finally {
...
} /** end of try construct
```

Assertions

- Statements in the program declaring a boolean expression regarding the current state of the computation
- When evaluated to true nothing happens
- When evaluated to false an AssertionError exception is thrown
- Can be disabled during runtime without program modification or recompilation
- Two forms
 - assert condition;
 - assert condition: expression;

Evaluation

- The types of exceptions makes more sense than in the case of C++
- The throws clause is better than that of C++ (The throw clause in C++ says little to the programmer)
- The finally clause is often useful
- The Java interpreter throws a variety of exceptions that can be handled by user programs

Summary of Exception Handling

- Ada provides extensive exception-handling facilities with a comprehensive set of built-in exceptions.
- C++ includes no predefined exceptions. Exceptions are bound to handlers by connecting the type of expression in the throw statement to that of the formal parameter of the catch function
- Java exceptions are similar to C++ exceptions except that a Java exception must be a descendant of the Throwable class. Additionally Java includes a finally clause

INTRODUCTION TO EVENT HANDLING

An event is a notification that something specific has occurred, such as a mouse click on a graphical button The event handler is a segment of code that is executed in response to an event

Java Swing GUI Components Text box is an object of class JTextField Radio button is an object of class JRadioButton Applet's display is a frame, a multilayered structure Content pane is one layer, where applets put output GUI components can be placed in a frame Layout manager objects are used to control the placement of components

The Java Event Model User interactions with GUI components create events that can be caught by event handlers, called event listeners An event generator tells a listener of an event by sending a message An interface is used to make event- handling methods conform to a standard protocol A class that implements a listener must implement an interface for the listener

One class of events is ItemEvent, which is associated with the event of clicking a checkbox, a radio button, or a list item The ItemListener interface prescribes a method, itemStateChanged, which is a handler for ItemEvent events The listener is created with addItemListener

Event Handling in C# & JAVA

Event Handling in C# (and the other.NET languages) is similar to that in Java.NET has two approaches, Windows Forms and Windows Presentation Foundation

We cover only the former (which is the original approach)

An application subclasses the Form predefined class (defined in System.Windows.Forms) There is no need to create a frame or panel in which to place the GUI components Label objects are used to place text in the window Radio buttons are objects of the RadioButton class

Components are positioned by assigning a new Point object to the Location property of the component `private RadioButton plain = new RadioButton(); plain.Location = new Point(100, 300); plain.Text = "Plain"; controls.Add(plain);`

All C# event handlers have the same protocol, the return type is void and the two parameters are of types object and EventArgs

An event handler can have any name A radio button is tested with the Boolean Checked property of the button `private void rb_CheckedChanged (object o, EventArgs e) { if (plain.Checked).....}`

To register an event, a new EventHandler object must be created and added to the predefined delegate for the event

When a radio button changes from unchecked to checked, the CheckedChanged event is raised

The associated delegate is referenced by the name of the event

If the handler was named `rb_CheckedChanged`, we could register it on the radio button named `plain` with: `plain.CheckedChanged += new EventHandler (rb_CheckedChanged);`

SUMMARY

Ada provides extensive exception-handling facilities with a comprehensive set of built-in exceptions.

C++ includes no predefined exceptions

Exceptions are bound to handlers by connecting the type of expression in the throw statement to that of the formal parameter of the catch function Java exceptions are similar to C++ exceptions except that a Java exception must be a descendant of the Throwable class.

Additionally, Java includes a finally clause.

An event is a notification that something has occurred that requires handling by an event handler Java event handling is defined on the Swing components C# event handling is the .NET model, which is similar to the Java model



UNIT-5

Functional Programming Languages & Scripting Language

5.1 Functional Programming Language Introduction - C05

- The design of the imperative languages is based directly on the von Neumann architecture
 - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on mathematical functions
 - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Mathematical Functions

- A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set
- A lambda expression specifies the parameter(s) and the mapping of a function in the following form

$$\lambda(x) x * x * x$$

for the function cube $(x) = x * x * x$

Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

e.g., $(\lambda(x) x * x * x)(2)$ which evaluates to 8

Functional Forms

- A higher-order function, or functional form, is one that either takes functions as parameters or yields a function as its result, or both

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

Form: $h \circ f \circ g$

which means $\lambda(x) h(x) \equiv f(g(x))$

For $f(x) = x + 2$ and $g(x) = 3 * x$, $h \circ f \circ g$ yields $(3 * x) + 2$

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

Form:

For $h(x)$

$(h, (2, 3, 4))$ yields (4, 9, 16)

5.2 Fundamentals of Functional Programming Languages- C05

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
 - In an imperative language, operations are done and the results are stored in variables for later use
 - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics

Referential Transparency

- In an FPL, the evaluation of a function always produces the same result given the same parameters

The First Functional Programming Language : LISP -

C05

LISP Data Types and Structures

- Data object types: originally only atoms and lists
 - List form: parenthesized collections of sublists and/or atoms
- e.g., (A B (C D) E)
- Originally, LISP was a typeless language
 - LISP lists are stored internally as single-linked lists

LISP Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.
- e.g., If the list (A B C) is interpreted as data it is a simple list of three atoms, A, B and C
If it is interpreted as a function application, it means that the function named A is applied to the two parameters, B and C
- The first LISP interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation

5.3 Support for Functional Programming in Primarily Imperative

Languages.ML - C05

- A static-scoped functional language with syntax that is closer to Pascal than to LISP
- Uses type declarations, but also does type inferencing to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

ML Specifics

- Function declaration form:

fun name (parameters) = body;

e.g., fun cube (x : int) = x * x * x;

- The type could be attached to return value, as in

fun cube (x) : int = x * x * x;

- With no type specified, it would default to int (the default for numeric values)

User-defined overloaded functions are not allowed, so if we wanted a cube function for real parameters, it would need to have a different name

- There are no type coercions in ML

- **ML selection**

if expression then then_expression

else else_expression

where the first expression must evaluate to a Boolean value

- Pattern matching is used to allow a function to operate on different parameter forms

```
fun fact(0) = 1
```

```
| fact(n : int) : int = n * fact(n - 1)
```

- Lists

Literal lists are specified in brackets

```
[3, 5, 7]
```

[] is the empty list CONS is the binary infix

operator, ::

4 :: [3, 5, 7], which evaluates to [4, 3, 5, 7]

CAR is the unary operator hd

CDR is the unary operator tl

```
fun length([]) = 0
```

```
| length(h :: t) = 1 + length(t);
```

```
fun append([], lis2) = lis2
```

```
| append(h :: t, lis2) = h :: append(t, lis2);
```

- The val statement binds a name to a value (similar to DEFINE in Scheme) val distance = time * speed;

- As is the case with DEFINE, val is nothing like an assignment statement in an imperative language

5.3 Haskell - C05

- Similar to ML (syntax, static scoped, strongly typed, type inferencing, pattern matching)
- Different from ML (and most other functional languages) in that it is purely functional (e.g., no variables, no assignment statements, and no side effects of any kind)

Syntax differences from ML

```
fact 0 = 1
```

```
fact n = n * fact (n - 1) fib 0 = 1
```

```
fib 1 = 1
```

```
fib (n + 2) = fib (n + 1) + fib n
```

Function Definitions with Different Parameter Ranges

```
fact n
```

```
| n == 0 = 1
```

```
| n > 0 = n * fact(n - 1)
```

```
sub n
```

```
| n < 10 = 0
```

```
| n > 100 = 2
```

```
| otherwise = 1 square x = x * x
```

- Works for any numeric type of x

Lists

- List notation: Put elements in brackets
e.g., directions = ["north", "south", "east", "west"]

- Length: #

e.g., #directions is 4

- Arithmetic series with the .. operator

e.g., [2, 4..10] is [2, 4, 6, 8, 10]

- Catenation is with ++

e.g., [1, 3] ++ [5, 7] results in [1, 3, 5, 7]

- CONS, CAR, CDR via the colon operator (as in Prolog)

e.g., 1:[3, 5, 7] results in [1, 3, 5, 7]

Factorial Revisited

product [] = 1

product (a:x) = a * product x

fact n = product

List Comprehension

- Set notation
- List of the squares of the first 20 positive integers: [n * n | n ← [1..20]]
- All of the factors of its given parameter:
factors n = [i | i ← [1..n div 2],
n mod i == 0]

Quicksort

sort [] = [] sort (a:x) =

sort [b | b ← x; b <= a] ++

[a] ++

sort [b | b ← x; b > a]

Lazy Evaluation

- A language is strict if it requires all actual parameters to be fully evaluated
- A language is nonstrict if it does not have the strict requirement
- Nonstrict languages are more efficient and allow some interesting capabilities
- infinite lists
- Lazy evaluation - Only compute those values that are necessary
- Positive numbers

positives = [0..]

- Determining if 16 is a square number

member [] b = False

member(a:x) b=(a == b)||member x b

squares = [n * n | n ← [0..]]

member squares

Member Revisited

- The member function could be written as: member [] b = False

member(a:x) b=(a == b)||member x b

- However, this would only work if the parameter to squares was a perfect square; if not, it will keep generating them forever. The following version will always work:

member2 (m:x) n

| m < n = member2 x n

| m == n = True

| otherwise = False

5.4 Applications of Functional Languages - CO5

- APL is used for throw-away programs
- LISP is used for artificial intelligence
- Knowledge representation
- Machine learning
- Natural language processing
- Modeling of speech and vision
- Scheme is used to teach introductory programming at some universities

Comparing Functional and Imperative Languages

- Imperative Languages:
 - Efficient execution
 - Complex semantics
 - Complex syntax
 - Concurrency is programmer designed
- Functional Languages:

- Simple semantics
- Simple syntax
- Inefficient execution
- Programs can automatically be made concurrent

Summary of Functional Programming Languages

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution instead of imperative features such as variables and assignments
- LISP began as a purely functional language and later included imperative features
- Scheme is a relatively simple dialect of LISP that uses static scoping exclusively
- COMMON LISP is a large LISP-based language
- ML is a static-scoped and strongly typed functional language which includes type inference, exception handling, and a variety of data structures and abstract data types
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- Purely functional languages have advantages over imperative alternatives, but their lower efficiency on existing machine architectures has prevented them from enjoying widespread use

Pragmatics

A software system often consists of a number of subsystems controlled or connected by a script. Scripting is a paradigm characterized by:

- Use of scripts to glue subsystems together.
- Rapid development and evolution of scripts.
- Modest efficiency requirements.
- Very high-level functionality in application-specific areas.

Key Concepts

The following concepts are characteristic of scripting languages:

- Very high-level string processing.
- Very high-level graphical user interface support.
- Dynamic typing.

Case Study: PYTHON

- PYTHON was designed in the early 1990s by Guido van Rossum.
- It has been used to help implement the successful Web search engine GOOGLE, and in a variety of other application areas ranging from science fiction (visual effects for the Star Wars series) to real science (computer-aided design in NASA).

Values and Types

- PYTHON has a limited repertoire of primitive types: integer, real, and complex numbers.
- It has no specific character type; single-character strings are used instead. Its boolean values (named False and True) are just small integers.
- PYTHON has a rich repertoire of composite types: tuples, strings, lists, dictionaries and objects. A PYTHON list is a heterogeneous sequence of values.
- A dictionary (sometimes called an associative array) is a heterogeneous mapping from keys to values, where the keys are distinct immutable values.
- The following code illustrates tuple construction:

```
date = 1998, "Nov", 19
```

Now `date[0]` yields 1998, `date[1]` yields "Nov", and `date[2]` yields 19.

- The following code illustrates two list constructions, which construct a homogeneous list and a heterogeneous list, respectively:

```
primes = [2, 3, 5, 7, 11]
```



```
years = ["unknown", 1314, 1707, date[0]]
```

Now `years[0]` yields 2, `years[1]` yields 1314, `years[3]` yields 1998, "`years[0] = 843`" updates the first component of `years`, and so on. Also, "`years.append(1999)`" adds 1999 at the end of `years`.

- The following code illustrates dictionary construction:

```
phones = {"David": 6742, "Carol": 6742, "Ali": 6046}
```

Now `phones["Carol"]` yields 6742, `phones["Ali"]` yields 6046, "`phones["Ali"] = 1234`" updates the component of `phones` whose key is "Ali", and so on. Also, "`David`" in `phones` returns True, and "`phones.keys()`" returns a list containing "Ali", "Carol", and "David" (in no particular order).

Variables, Storage and Control

- PYTHON supports global and local variables.
- Variables are not explicitly declared, simply initialized by assignment. After initialization, a variable may later be assigned any value of any type.
- PYTHON's repertoire of commands include assignments, procedure calls, conditional (if-but not case-) commands, iterative (while- and for-) commands and exception-handling commands.

- However, PYTHON differs from C in not allowing an assignment to be used as an expression.

- PYTHON additionally supports simultaneous assignment.

- For example:

```
y, m, d = date
```

assigns the three components of the tuple `date` to three separate variables. Also:

```
m, n = n, m
```

concisely swaps the values of two variables `m` and `n`. (Actually, it first constructs a pair, then assigns the two components of the pair to the two left- side variables)

- PYTHON if- and while-commands are conventional.
- PYTHON for-commands support definite iteration.
- We can easily achieve the conventional iteration over a sequence of numbers by using the library procedure `range(m,n)`, which returns a list of integers from `m` through `n-1`.

- PYTHON supports break, continue, and return sequencers. It also supports exceptions, which are objects of a subclass of Exception, and which can carry values.

- The following code computes the Greatest Common Divisor of two integers, `m` and `n`:

```
p, q = m, n
```

```
while p % q != 0:
```

```
    p, q = q, p % q
gcd = q
```

- Note the elegance of simultaneous assignment.
- Note also that indentation is required to indicate the extent of the loop body.
- The following code sums the numeric components of a list `row`, ignoring any nonnumeric components:

```
sum = 0.0
```

```
for x in row:
```

```
    if isinstance(x, (int, float)): sum += x
```

PYTHON Exceptions

- The following code prompts the user to enter a numeric literal, and stores the corresponding real number in `num`:

```
while True:
```

```
    try:
```

```
        response = raw_input("Enter a numeric literal: ")
        num = float(response)
```



```
break
except ValueError:
print "Your response was ill-formed."
```

This while-command keeps prompting until the user enters a well-formed numeric literal. The library procedure `raw_input(...)` displays the given prompt and returns the user's response as a string. The type conversion "`float(response)`" attempts to convert the response to a real number. If this type conversion is possible, the following break sequencer terminates the loop. If not, the type conversion throws a `ValueError` exception, control is transferred to the `ValueError` exception handler, which displays a warning message, and finally the loop is iterated again.

Bindings and Scope

- A PYTHON program consists of a number of modules, which may be grouped into packages.
- Within a module we may initialize variables, define procedures, and declare classes. Within a procedure we may initialize local variables and define local procedures. Within a class we may initialize variable components and define procedures (methods).
- During a PYTHON session, we may interactively issue declarations, commands, and expressions from the keyboard.
- These are all acted upon immediately. Whenever we issue an expression, its value is displayed on the screen. We may also import a named module (or selected components of it) at any time.
- PYTHON was originally a dynamically-scoped language, but it is now statically scoped.

Procedural Abstraction

- PYTHON supports function procedures and proper procedures.
- The only difference is that a function procedure returns a value, while a proper procedure returns nothing.
- Since PYTHON is dynamically typed, a procedure definition states the name but not the type of each formal parameter. The corresponding argument may be of different types on different calls to the procedure.

PYTHON Procedures

- The following function procedure returns the greatest common divisor of its two arguments:

```
def gcd (m, n):
    p, q = m, n
    while p % q != 0:
        p, q = q, p % q
    return q
```

Here p and q are local variables.

- The following proper procedure takes a date represented by a triple and prints that date in ISO format (e.g., "2000-01-01"):

```
def print_date (date):
    y, m, d = date
    if m = "Jan":
        m = 1
    elif m = "Feb":
        m = 2
    ...
    elif m = "Dec":
        m = 12
    print "%04d-%02d-%02d" % (y, m, d)
```

Here y, m, and d are local variables.

PYTHON procedure with dynamic typing

- The following function procedure illustrates the flexibility of dynamic typing. It returns the minimum and maximum component of a given sequence:

```
def minimax (vals):  
    min = max = vals[0]  
    for val in vals:  
        if val < min:  
            min = val  
        elif val > max:  
            max = val  
    return min, max
```



- In a call to this procedure, the argument may be either a tuple or a list.
- In effect it has two results, which we can easily separate using simultaneous assignment:
`readings = [. .]`
`low, high = minimax(readings)`
- Some older languages such as C have library procedures with variable numbers of arguments.
- PYTHON is almost unique in allowing such procedures to be defined by programmers.
- This is achieved by the simple expedient of allowing a single formal parameter to refer to a whole tuple (or dictionary) of arguments.

PYTHON procedure with a variable number of arguments

- The following proper procedure accepts any number of arguments, and prints them one per line:

```
def printall (*args):
for arg in args:
print arg
```

- The notation “*args” declares that args will refer to a tuple of arguments.
- All of the following procedure calls work successfully:
`printall(name)` `printall(name, address)`
`printall(name, address, zipcode)`

Data Abstraction

- PYTHON has three different constructs relevant to data abstraction: packages, modules, and classes.
- Modules and classes support encapsulation, using a naming convention to distinguish between public and private components.
- A package is simply a group of modules. A module is a group of components that may be variables, procedures, and classes.
- These components may be imported for use by any other module. All components of a module are public, except those whose identifiers start with “_” which are private.
- A class is a group of components that may be class variables, class methods, and instance methods. A procedure defined in a class declaration acts as an instance method if its first formal parameter is named self and refers to an object of the class being declared. Otherwise the procedure acts as a class method.
- To achieve the effect of a constructor, we usually equip each class with an initialization method named “init”; this method is automatically called when an object of the class is constructed. Instance variables are named using the usual “.” Notation (as in self.attr), and they may be initialized by the initialization method or by any other method. All components of a class are public, except those whose identifiers start with “_”, which are private.

PYTHON Class

- Consider the following class:

class Person:

```
def __init__(self, sname, fname,
gender, birth):self._surname =
sname
self._forename = fname
self._female = (gender == "F" or
gender == "f")self._birth = birth
def get_surname (self):
return self._surname
def change_surname
(self, sname):self._
surname = sname
def print_details (self):
print self._forename + " " + self._surname
```

- This class is equipped with an initialization method and three other instance methods, each of which has a self parameter and perhaps some other parameters. In the following code:

```
dw = Person("Watt", "David", "M", 1946)
```

the object construction on the right first creates an object of class Person; it then passes the above arguments, together with a reference to the newly created object, to the initialization method. The latter initializes the object's instance variables, which are named surname, forename, female, and birth (and thus are all private).

- PYTHON supports multiple inheritance: a class may designate any number of superclasses. Ambiguous references to class components are resolved by searching the superclasses in the order in which they are named in the class declaration.
- PYTHON's support for object-oriented programming is developing but is not yet mature. The use of the “ ” naming convention to indicate privacy is clumsy and error-prone; class components are public by default. Still more seriously, variable components can be created (and deleted) at any time, by any method and even by application code.

Separate Compilation

- PYTHON modules are compiled separately. Each module must explicitly import every other module on which it depends. Each module's source code is stored in a text file.
- For example, a module named widget is stored in a file named widget.py. When that module is first imported, it is compiled and its object code is stored in a file named widget.pyc.
- Whenever the module is subsequently imported, it is recompiled only if the source code has been edited in the meantime. Compilation is completely automatic.
- The PYTHON compiler does not reject code that refers to undeclared identifiers. Such code simply fails if and when it is executed.

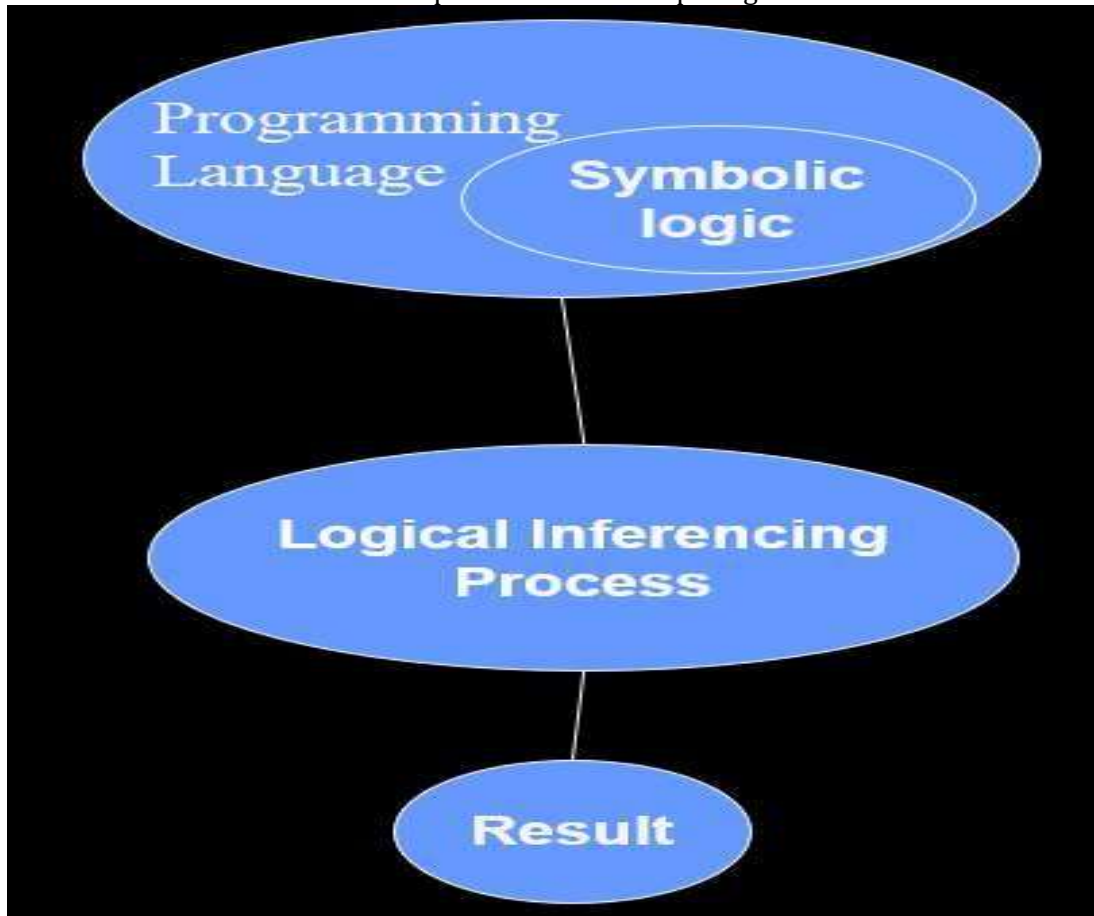
Module Library

- PYTHON is equipped with a very rich module library, which supports string handling, markup, mathematics and cryptography, multimedia, GUIs, operating system services, Internet services, compilation, and so on.
- Unlike older scripting languages, PYTHON does not have built-in high-level string processing or GUI support. Instead, the PYTHON module library provides such functionality. For example, the re library module provides powerful string matching facilities using regular expressions.
- The pragmatic issues that influence the design of scripting languages: gluing, rapid development and evolution, modest efficiency requirements, and very high-level functionality in relevant areas.
- The concepts common to most scripting languages: very high-level support for string processing, very high-level support for GUIs, and dynamic typing.
- The design of a major scripting language PYTHON, resembles a conventional programming language, except that it is dynamically typed, and that it derives much of its expressiveness from a rich module library.

Logic Programming Language

- To introduce the concepts of logic programming and logic programming languages

- To introduce a brief description of a subset of prolog



- Logic programs are declarative programs
 - Specify the desired results - true
 - State the fact
- The major difference between logic programming and other programming languages (imperative and functional)
 - Every data item that exist in logic programming has written in specific representation (symbolic logic)
- Prolog is a logic programming that widely used logic language