CONTENTS

UNIT-I

UNIT-II

UNIT-III

1. Fundamentals of sub-programs
2. Scope and lifetime of variable
3. Static and dynamic scope
4. Design issues of subprograms and operations, local referencing environments
5. Parameter passing methods
6. Overloaded sub-programs
7. Generic sub-programs
8. Parameters that are sub-program names
9. Design issues for functions user defined overloaded operators, co routines.

UNIT-IV

1. Abstract Data types
2. Concurrency
3. Exception handling
4. Logic Programming Language

UNIT-V

1. Functional Programming Languages
2. Introduction
3. LISP, ML, Haskell
4. Scripting Language: Pragmatics
5. Python
6. Procedural abstraction, data abstraction, separate compilation, module library

UNIT – I

PRELIMINARY CONCEPTS

Reasons for Studying Concepts of Programming Languages

- **Increased ability to express ideas.**
    - It is believed that the depth at which we think is influenced by the expressive power of the language in which we communicate our thoughts. It is difficult for people to conceptualize structures they can't describe, verbally or in writing.
    - Language in which they develop S/W places limits on the kinds of control structures, data structures, and abstractions they can use.
    - Awareness of a wider variety of P/L features can reduce such limitations in S/W development.
    - Can language constructs be simulated in other languages that do not support those constructs directly?

- **Improved background for choosing appropriate languages**
    - Many programmers, when given a choice of languages for a new project, continue to use the language with which they are most familiar, even if it is poorly suited to new projects.
    - If these programmers were familiar with other languages available, they would be in a better position to make informed language choices.

- **Greater ability to learn new languages**
    - Programming languages are still in a state of continuous evolution, which means continuous learning is essential.
    - Programmers who understand the concept of OO programming will have easier time learning Java.

- Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes easier to see how concepts are incorporated into the design of the language being learned.

- **Understand significance of implementation**
  - Understanding of implementation issues leads to an understanding of why languages are designed the way they are.
  - This in turn leads to the ability to use a language more intelligently, as it was designed to be used.

- **Ability to design new languages**
  - The more languages you gain knowledge of, the better understanding of programming languages concepts you understand.

- **Overall advancement of computing**
  - In some cases, a language became widely used, at least in part, b/c those in positions to choose languages were not sufficiently familiar with P/L concepts.
  - Many believe that ALGOL 60 was a better language than Fortran; however, Fortran was most widely used. It is attributed to the fact that the programmers and managers didn't understand the conceptual design of ALGOL 60.
  - Do you think IBM has something to do with it?

Programming Domains

- **Scientific applications**
  - In the early 40s computers were invented for scientific applications.
  - The applications require large number of floating point computations.
  - Fortran was the first language developed scientific applications.
  - ALGOL 60 was intended for the same use.
- **Business applications**
  - The first successful language for business was COBOL.
  - Produce reports, use decimal arithmetic numbers and characters.
  - The arrival of PCs started new ways for businesses to use computers.

- Spreadsheets and database systems were developed for business.

- **Artificial intelligence**
  - Symbolic rather than numeric computations are manipulated.
  - Symbolic computation is more suitably done with linked lists than arrays.
  - LISP was the first widely used AI programming language.
- **Systems programming**
  - The O/S and all of the programming supports tools are collectively known as its system software.
  - Need efficiency because of continuous use.
- **Scripting languages**
  - Put a list of commands, called a script, in a file to be executed.
  - PHP is a scripting language used on Web server systems. Its code is embedded in HTML documents. The code is interpreted on the server before the document is sent to a requesting browser.
- Special-purpose languages

Language Evaluation Criteria

Readability

- Software development was largely thought of in term of writing code "**LOC**".
- Language constructs were designed more from the point of view of the computer than the users.
- Because ease of maintenance is determined in large part by the readability of programs, readability became an important measure of the quality of programs and programming languages. The result is a crossover from focus on machine orientation to focus on human orientation.
- The most important criterion "ease of use"
- **Overall simplicity** "Strongly affects readability"
  - Too many features make the language difficult to learn. Programmers tend to learn a subset of the language and ignore its other features. "ALGOL 60"
  - Multiplicity of features is also a complicating characteristic "having more than one way to accomplish a particular operation."
  - Ex "**Java**":

```
count = count + 1
```

count

+= 1

count

++

++count

- Although the last two statements have slightly different meaning from each other and from the others, all four have the same meaning when used as stand-alone expressions.
- Operator overloading where a single operator symbol has more than one meaning.
- Although this is a useful feature, it can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly.

- **Orthogonality**
  - Makes the language easy to learn and read.
  - Meaning is context independent. Pointers should be able to point to any type of variable or data structure. The lack of orthogonality leads to exceptions to the rules of the language.
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.
  - Every possible combination is legal and meaningful.
  - Ex: page 11 in book.
  - The more orthogonal the design of a language, the fewer exceptions the language rules require.
  - The most orthogonal programming language is ALGOL 68. Every language construct has a type, and there are no restrictions on those types.
  - This form of orthogonality leads to unnecessary complexity.

- **Control Statements**
  - It became widely recognized that indiscriminate use of goto statements severely reduced program readability.
  - Ex: Consider the following nested loops written
    in C while (incr < 20)

{

```
while (sum <= 100
  {
      sum += incr;
   }
   incr++;
}
```

if C didn't have a loop construct, this would be written as follows:

```
loop1:
  if (incr >= 20) go to out;
loop2:
  if (sum > 100) go to
    next; sum += incr;
    go to
loop2; next:
  incr++;
  go to
loop1:
out:
```

- Basic and Fortran in the early 70s lacked the control statements that allow strong restrictions on the use of gotos, so writing highly readable programs in those languages was difficult.
- Since then, languages have included sufficient control structures.
- The control statement design of a language is now a less important factor in readability than it was in the past.

- **Data Types and Structures**
  - The presence of adequate facilities for defining data types and data structures in a language is another significant aid to reliability.
  - Ex: Boolean type.
    timeout = 1 or timeout = true

- **Syntax Considerations**
  - The syntax of the elements of a language has a significant effect on readability.
  - The following are examples of syntactic design choices that affect readability:
    - ***Identifier forms***: Restricting identifiers to very short lengths detracts from readability. ANSI BASIC (1978) an identifier could consist only of a single letter of a single letter followed by a single digit.
    - ***Special Words***: Program appearance and thus program readability are strongly influenced by the forms of a language's special words. Ex: **while**, **class**, **for**. C uses braces for pairing control structures. It is difficult to determine which group is being ended. Fortran 95 allows programmers to use special names as legal variable names.
    - ***Form and Meaning***: Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability.
    - Semantic should follow directly from syntax, or form.
    - Ex: In C the use of **static** depends on the context of its appearance.
      If used as a variable inside a function, it means the variable is created at compile time.

      If used on the definition of a variable that is outside all functions, it means the variable is visible only in the file in which its definition appears.

Writability

- It is a measure of how easily a language can be used to create programs for a chosen problem domain.
- Most of the language characteristics that affect readability also affect writability.
- Simplicity and orthogonality

- A smaller number of primitive constructs and a consistent set of rules for combining them is much better than simply having a large number of primitives.

- Support for abstraction
  - **Abstraction** means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.
  - A process abstraction is the use of a subprogram to implement a sort algorithm that is required several times in a program instead of replicating it in all places where it is needed.
- Expressivity
  - It means that a language has relatively convenient, rather than cumbersome, ways of specifying computations.
  - Ex: ++count ⇔ count = count + 1 // more convenient and shorter

Reliability

- A program is said to be **reliable** if it performs to its specifications under all conditions.
- **Type checking**: is simply testing for type errors in a given program, either by the compiler or during program execution.
  - The earlier errors are detected, the less expensive it is to make the required repairs. Java requires type checking of nearly all variables and expressions at compile time.
- **Exception handling**: the ability to intercept run-time errors, take corrective measures, and then continue is a great aid to reliability.
- **Aliasing**: it is having two or more distinct referencing methods, or names, for the same memory cell.
  - It is now widely accepted that aliasing is a dangerous feature in a language.
- **Readability and writability**: Both readability and writability influence reliability.

Cost

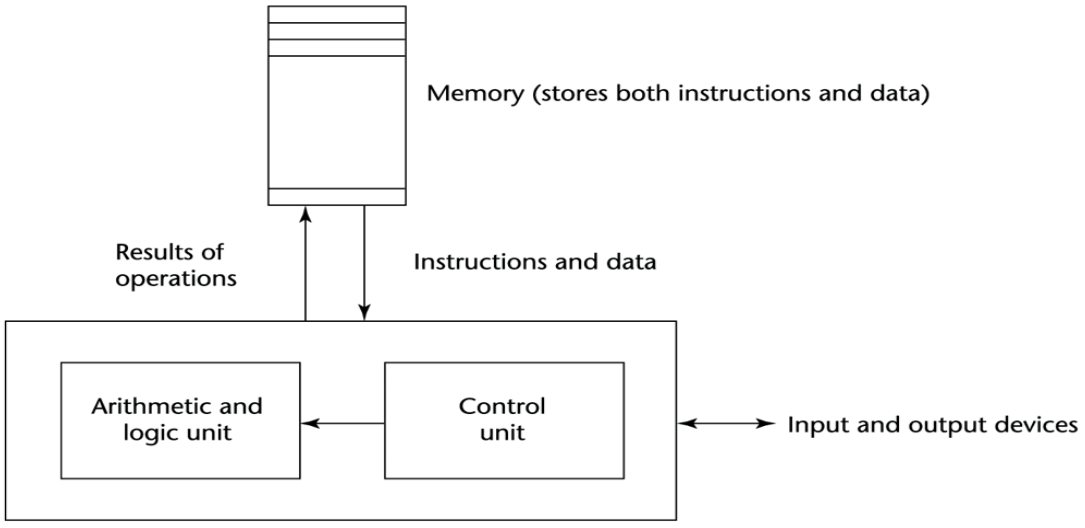
- Categories
  - Training programmers to use language
  - **Writing programs** "Writability"
  - Compiling programs
  - Executing programs

- Language implementation system "Free compilers is the key, success of Java"
- **Reliability**, does the software fail?
- **Maintaining** programs: Maintenance costs can be as high as two to four times as much as development costs.
- Portability "standardization of the language" ,Generality (the applicability to a wide range of applications)

Influences on Language Design

Computer architecture: Von Neumann

- We use imperative languages, at least in part, because we use von Neumann machines
  - Data and programs stored in same memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Results of operations in the CPU must be moved back to memory
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient



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

Language Categories

- **Imperative**
  - Central features are variables, assignment statements, and iteration
  - C, Pascal
- **Functional**
  - Main means of making computations is by applying functions to given parameters
  - LISP, Scheme
- **Logic**
  - Rule-based
  - Rules are specified in no special order
  - Prolog
- **Object-oriented**
  - Encapsulate data objects with processing
  - Inheritance and dynamic type binding
  - Grew out of imperative languages
  - C++, Java
  -

Language Design Trade-Offs

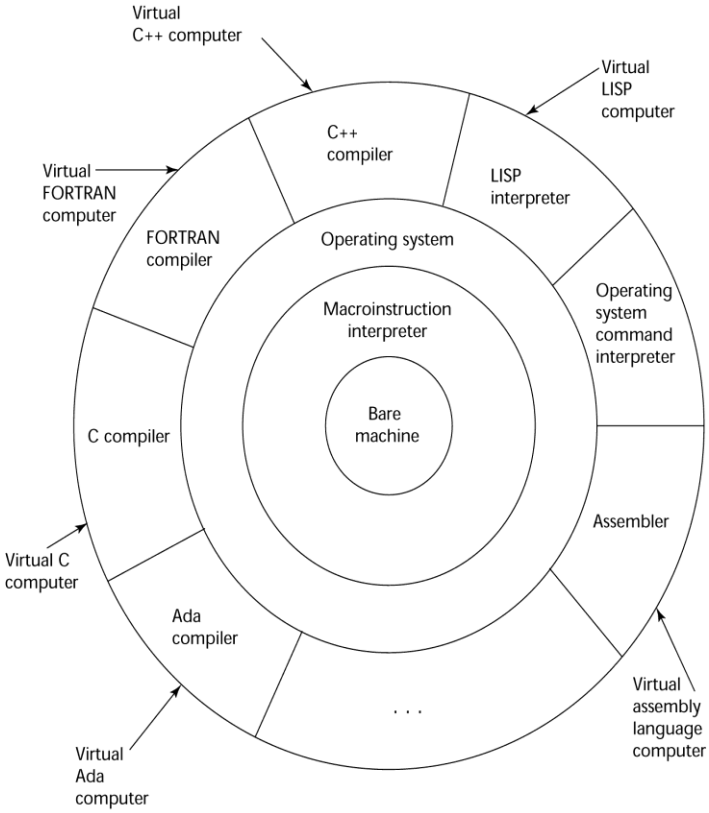
- Reliability vs. cost of execution – Conflicting criteria – Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability – Another conflicting criteria – Example: APL provides many powerful operators (and 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

1.1 Implementation Methods

- **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

Layered View of Computer

The operating system and language implementation are layered over Machine interface of a computer



Compilation

- Translate high-level program (source language) into machine code (machinelanguage)
- Slow translation, fast execution
- Compilation process has several phases:

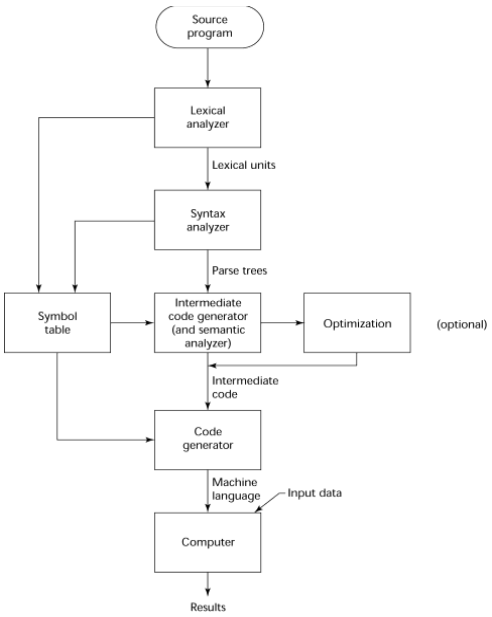
lexical analysis: converts characters in the source program into lexicalunits

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

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

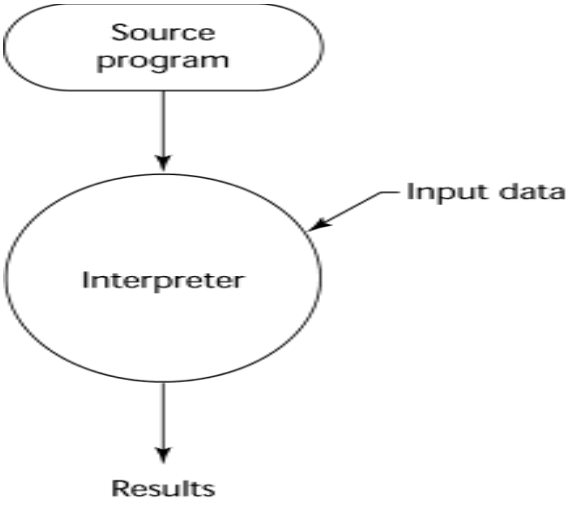
- Fetch-execute-cycle (on a von Neumann architecture) initialize the program counter repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
  end repeat

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

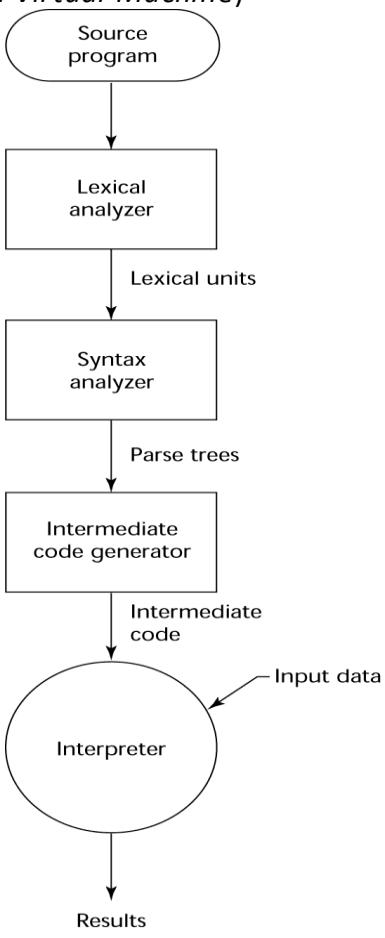
Pure Interpretation

- 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
- Significant comeback with some 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 thatallows 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 run- time system (together, these are called *Java Virtual Machine*)



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

Preprocessors

- 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

Syntax and Semantics

Syntax - the form or structure of the expressions, statements, and program units

Semantics - the meaning of the expressions, statements, and program units

Who must use language definitions?

1. Other language designers
2. Implementors
3. Programmers (the users of the language)

A **sentence** is a string of characters over some alphabetA

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)

Formal approaches to describing syntax:

Recognizers - used in compilers

Generators - what we'll study

Language recognizers:

Suppose we have a language L that uses an alphabet ∑ of characters. To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet ∑. R would indicate whether a given input string was or was not in L. In effect, R would either accept or reject the given string. Such devices are like filters, separating legal sentences from those that are incorrectly formed. If R, when fed any string of characters over ∑, accepts it only if it is in L, then R is a description of L. Because most useful languages are, for allpractical purposes, infinite, this might seem like a lengthy and ineffective process. Recognition devices, however, are not used to

enumerate all of the sentences of a language—they have a different purpose.

The syntax analysis part of a compiler is a recognizer for the language the compiler translates. In this role, the recognizer need not test all possible strings of characters from some set to determine whether each is in the language.

Language Generators

A language generator is a device that can be used to generate the sentences of a language. The syntax-checking portion of a compiler (a language recognizer) is not as useful a language description for a programmer because it can be used only in trial-and-error mode. For example, to determine the correct syntax of a particular statement using a compiler, the programmer can only submit a speculated version and note whether the compiler accepts it. On the other hand, it is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator.

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.

A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminal* and *nonterminal* symbols

BNF:

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

Syntactic lists are described in BNF 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

An example derivation:

<program> => <stmts> => <stmt>

=> <var> = <expr> => a = <expr>

=> a = <term> + <term>

=> a = <var> + <term>

=> a = b + <term>

=> a = b + const

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 non terminal in each sentential formis the one that is expanded A derivation may be neither leftmost nor rightmost

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 An

example derivation:

<program> => <stmts> => <stmt>

=> <var> = <expr> => a = <expr>

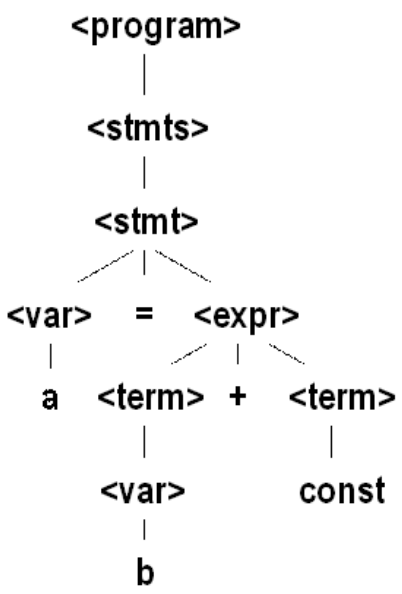
=> a = <term> + <term>

=> a = <var> + <term>

=> a = b + <term>

=> a = b + const

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 A *parse tree* is a hierarchical representation of a derivation



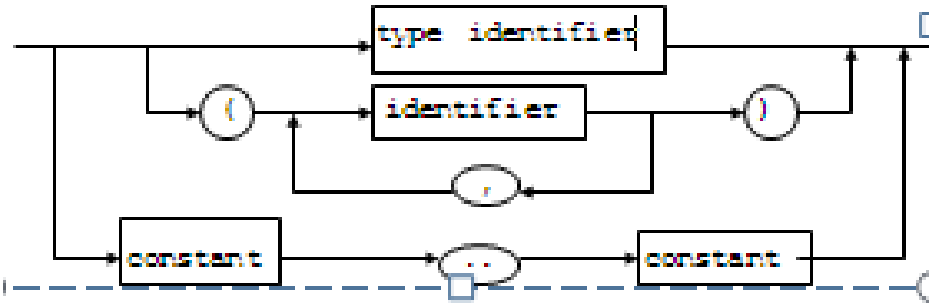
EBNF:

<expr> -> <term> {(+ | -) <term>}

<term> -> <factor> {(* | /) <factor>}

Syntax Graphs - put the terminals in circles or ellipses and put the nonterminals in rectangles; connect with lines with arrowheads

e.g., Pascal type declarations



Static semantics (have nothing to do with meaning)

Categories:

1. **Context-free** but cumbersome (e.g. type checking)
2. **Noncontext-free** (e.g. variables must be declared before they are used

Attribute Grammars (AGs) (Knuth, 1968) Cfgs cannot describe all of the syntax of programming languages

- Additions to cfgs to carry some semantic info along through parse tree Primary value of AGs:

1. Static semantics specification
2. Compiler design (static semantics checking)

Def: An *attribute grammar* is a cfg G = (S, N, T, P) with the following additions:

1. For each grammar symbol x there is a set A(x) of attribute values
2. Each rule has a set of functions that define certain attributes of the nonterminals in the rule
3. Each rule has a (possibly empty) set of predicates to check for attribute consistency

Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule Functions of the form $S(X_0) = f(A(X_1), \ldots A(X_n))$ define *synthesized attributes* Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, for $i <= j <= n$, define *inherited attributes* Initially, there are *intrinsic attributes* on the leaves

Example*:* expressions of the form id + id - id's can be either int_type or real_type

types of the two id's must be the same type of the expression must match it's expected type

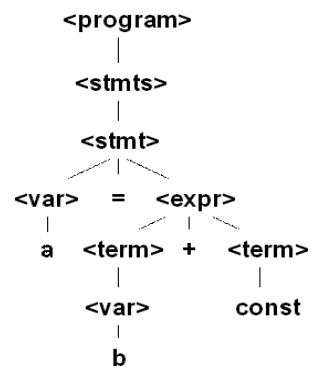
BNF:

<expr> -> <var> + <var>

<var> -> id

PARSE TREE:

A hierarchical representation of a derivation. Every internal node of a parse tree is labeled with a non terminal symbol; every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence



Ambiguous grammar

A grammar is **ambiguous** if it generates a sentential form that has two or more distinct parse trees An ambiguous expression grammar:

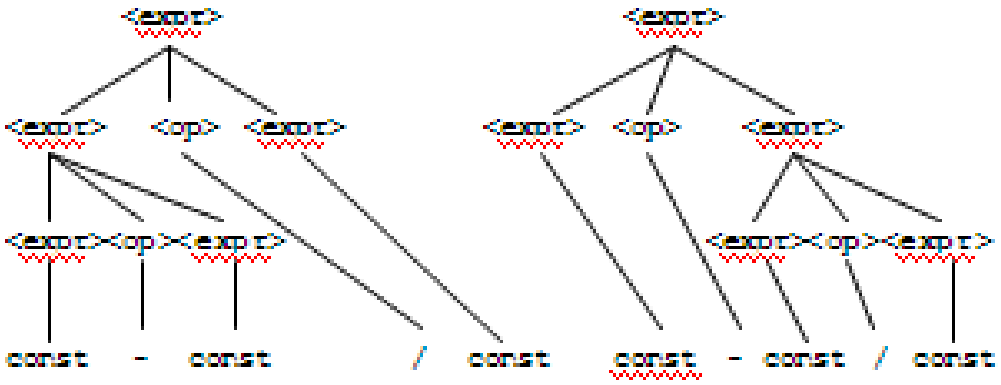
<expr> -> <expr> <op> <expr> | const

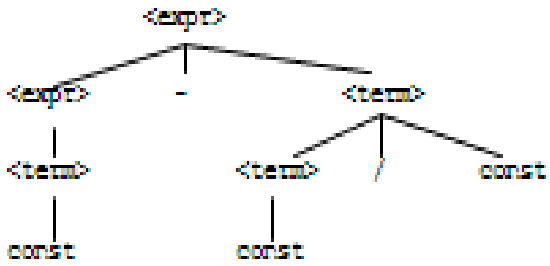
<op> -> / | -

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity An unambiguous expression grammar:

<expr> -> <expr> - <term> | <term>

<term> -> <term> / const | const





expr> => <expr> - <term> => <term> - <term>

=> const - <term>

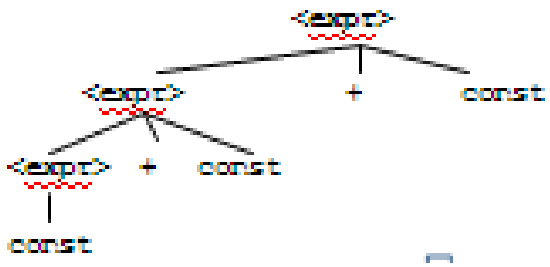
=> const - <term> / const

=> const - const / const

Operator associativity can also be indicated by a grammar

<expr> -> <expr> + <expr> | const (ambiguous)

<expr> -> <expr> + const | const (unambiguous)



Extended BNF (just abbreviations):

1. Optional parts are placed in brackets ([]) <proc_call> -> ident [(<expr_list>)]
2. Put alternative parts of RHSs in parentheses and separate them with vertical bars <term> -> <term> (+ | -) const
3. Put repetitions (0 or more) in braces ({ }) <ident> -> letter {letter | digit}

BNF:

<expr> -> <expr> + <term>

| <expr> - <term>

| <term>

<term> -> <term> * <factor>

| <term> / <factor>

| <factor>

Data types

Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types
- Names
- Variables
- The Concept of Binding
- Type Checking
- Strong Typing
- Type Compatibility
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- A *data type* defines a collection of data objects and a set of predefinedoperations 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 arethey specified?

Primitive Data Types

- 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

Primitive Data Types: 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

Primitive Data Types: 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-PointStandard 754

Primitive Data Types: 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

Primitive Data Types: 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

Primitive Data Types: 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

Primitive Data Types: Character

- Stored as numeric codings
- 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?

Character String Types 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

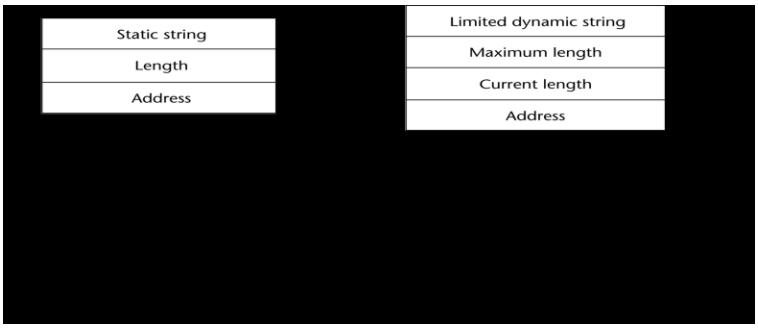
Character String Type Evaluation

- 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?

Character String 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

Compile- and Run-Time Descriptors



User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easilyassociated 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 onlycertain range of values

- Reliability

Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementation of User-Defined Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (bythe 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 thefirst 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 specifyrange checking
- Java, ML, C# specify range checking
- In Ada, the default is to require rangechecking, 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 thearray 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)

Subscript Binding and Array Categories (continued)

- 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 programexecution)

- 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# exampleint
  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 andmatrixes 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. Pythonalso supports array catenation and element membership operations
- Ruby also provides array catenation
- Fortran provides *elemental* operations because they are between pairs ofarray elements

  —For example, + operator between two arrays results in an array of the sums ofthe element pairs of the two arrays

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rowshave 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 jaggedarrays)

Slices

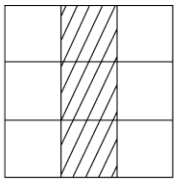
- A slice is some substructure of an array; nothing more than a referencingmechanism
- Slices are only useful in languages that have array operations

Slice Examples

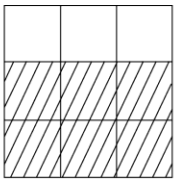
- Fortran 95

Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
Vector (3:6) is a four element array**Slices**

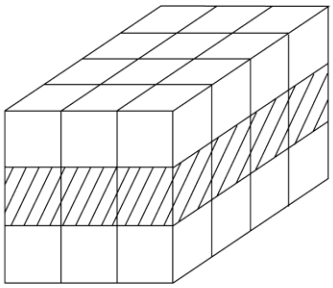
Examples in Fortran 95



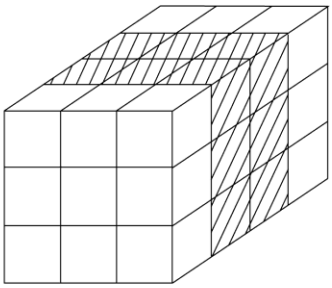
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

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)

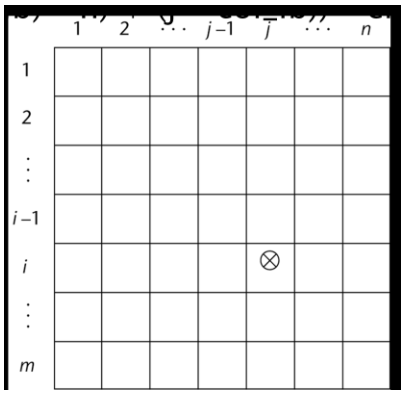
Accessing Multi-dimensioned Arrays

- Two common ways:

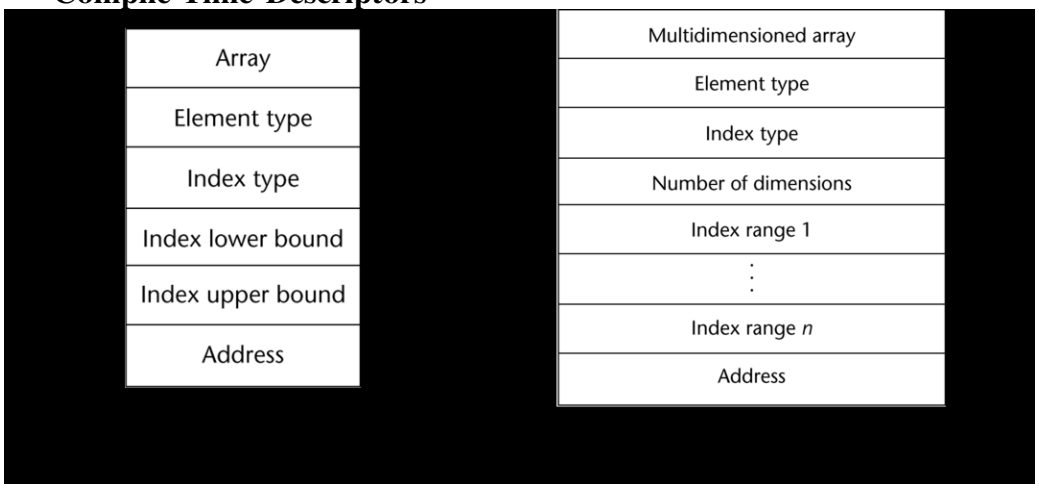
–Row major order (by rows) – used in most languages

–column major order (by columns) – used in Fortran

Locating an Element in a Multi-dimensioned Array



Compile-Time Descriptors



Associative Arrays

- An *associative array* is an unordered collection of data elements that areindexed 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 **%; l**iterals 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 deletedelete

```
$hi_temps{"Tue"};
```

Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which theindividual 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 recursivedefinition

```
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 recordFirst: 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

1. COBOL

field_name OF record_name_1 OF ... OF record_name_n

2. 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 referenceis unambiguous, for example in COBOL

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

Operations on Records

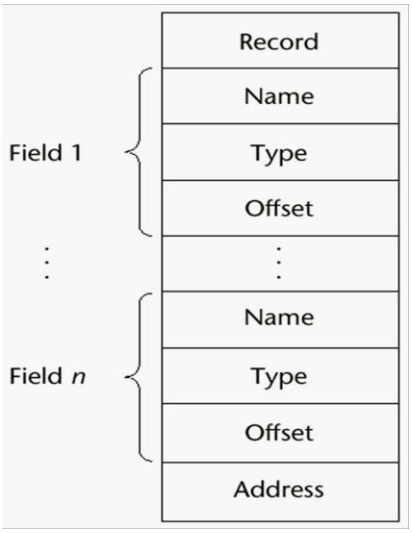
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING

  Copies a field of the source record to the corresponding field in the targetrecord

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, becausesubscripts are dynamic (field names are static)
  - Dynamic subscripts could be used with record field access, but it wouldddisallow type checking and it would be much slower

Implementation of Record Type



Unions Types

- A *union* is a type whose variables are allowed to store different type values atdifferent 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 languagesupport for type checking; the union in these languages is called *free union*

- Type checking of unions require that each union include a type indicatorcalled a *discriminant*

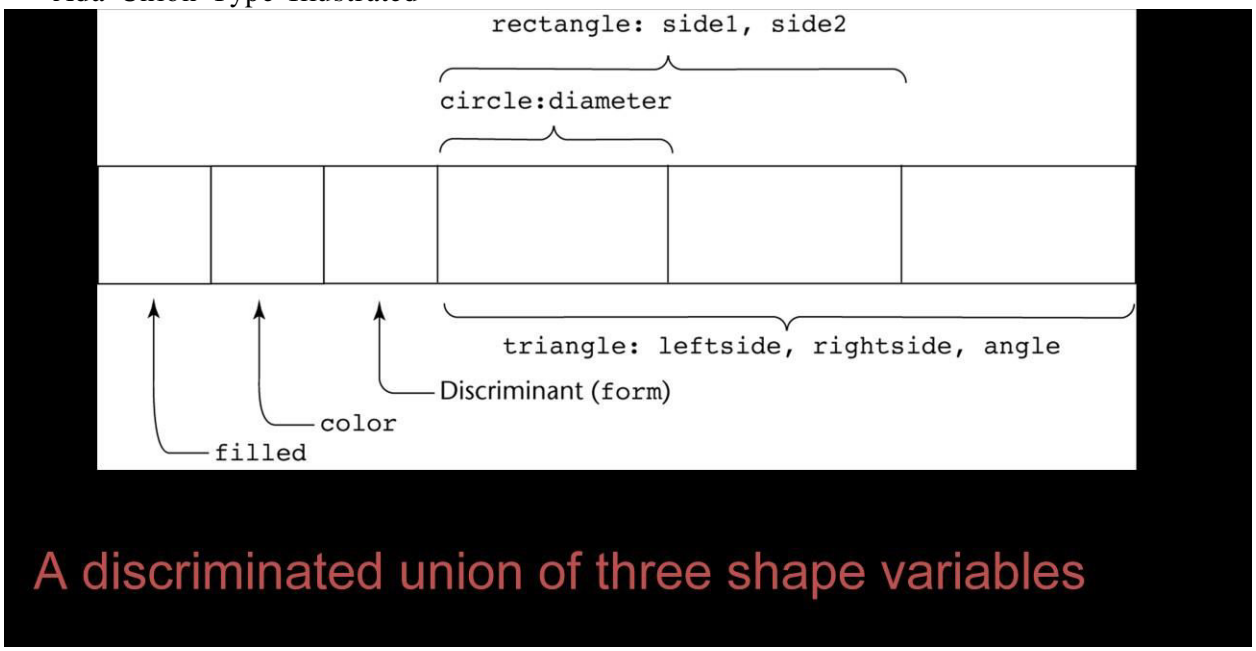
–Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);type Colors is
    (Red, Green, Blue);
type Figure (Form: Shape) is recordFilled:
    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;

Ada Union Type Illustrated



Evaluation of Unions

- Free unions are unsafe

–Do not allow type checking

- Java and C# do not support unions

–Reflective of growing concerns for safety in programming language

- Ada‘s descriminated unions are safe

Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memoryaddresses 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 isdynamically 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, orboth?
- Should the language support pointer types, reference types, or both?

Pointer Operations

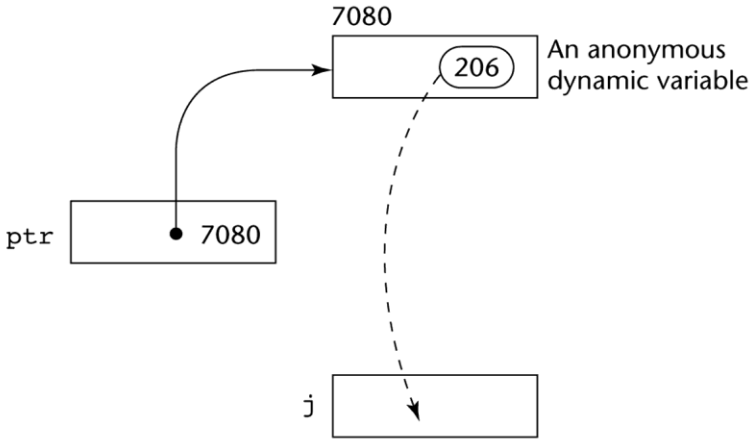
- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable‘s value to some useful address
- Dereferencing yields the value stored at the location represented by thepointer‘s value

–Dereferencing can be explicit or implicitC++ uses an explicit operation via *

```
j = *ptr
```

sets j to the value located at ptr

Pointer Assignment Illustrated



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-dynamicvariable
- The process of losing heap-dynamic variables is called *memory leakage*

Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can beautomatically deallocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada (possiblewith 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 wasallocated
- 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 isused 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 bya variable
  - Pointers or references are necessary for dynamic data structures--so we can'tdesign 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 lockvalue

–When heap-dynamic variable allocated, lock value is created and placed inlock 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 variablespace becomes empty

Reference Counter

- Reference counters: maintain a counter in every cell that store the number ofpointers currently pointing at the cell

~~Di~~sadvantages: space required, execution time required, complications forcells connected circularly

~~Ad~~vantage: it is intrinsically incremental, so significant delays in theapplication execution are avoided

Mark-Sweep

- The run-time system allocates storage cells as requested and disconnectspointers from cells as necessary; mark-sweep then begins

–Every heap cell has an extra bit used by collection algorithm

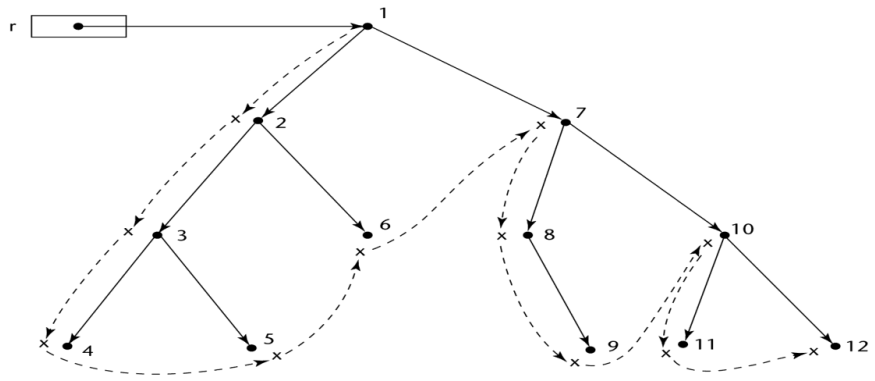
–All cells initially set to garbage

–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 mark- sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Marking Algorithm



Dashed lines show the order of node_marking

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 in nontrivial

–Maintaining the list of available space is another source of overhead

Names

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

- **Length**
- 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 implementors often impose one
- **Connectors**
- Pascal, Modula-2, and FORTRAN 77 don't allow
- Others do
- Case sensitivity
- Disadvantage: readability (names that look alike are different)
- worse in C++ and Java because predefined names are mixed case (e.g.
- IndexOutOfBoundsException)
- C, C++, and Java names are case sensitive
- The names in other languages are not
- Special words
- An aid to readability; used to delimit or separate statement clauses
- Def: A keyword is a word that is special only in certain contexts i.e. in Fortran: Real VarName (*Real is data type followed with a name, therefore Real is a keyword)* Real = 3.4 (*Real is a variable)*
- Disadvantage: poor readability
- Def: A reserved word is a special word that cannot be used as a user-definedname

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:(name, address, value, type, lifetime, and scope)
- Name - not all variables have them (anonymous)
- Address - the memory address with which it is associated (also called *l*-value)
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program
  - If two variable names can be used to access the same memory location, they arecalled aliases
  - Aliases are harmful to readability (program readers must remember all of them)
- How aliases can be created:
  - Pointers, reference variables, C and C++ unions, (and through parameters -discussed in Chapter 9)
  - Some of the original justifications for aliases are no longer valid; e.g. memory
    - reuse in FORTRAN
  - Replace them with dynamic allocation
- 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 theprecision
- Value - the contents of the location with which the variable is associated
- Abstract memory cell - the physical cell or collection of cells associated with avariable

The Concept of Binding

- The *l*-value of a variable is its address

- The *r*-value of a variable is its value
- Def: A binding is an association, such as between an attribute and an entity, orbetween an operation and a symbol
- Def: Binding time is the time at which a binding takes place.
- Possible binding times:
    - Language design time--e.g., bind operator symbols to operations
    - Language implementation time--e.g., bind floating point type to a representation
    - Compile time--e.g., bind a variable to a type in C or Java
    - Load time--e.g., bind a FORTRAN 77 variable to a memory cell (or a C **static**
- variable)
    - Runtime--e.g., bind a nonstatic local variable to a memory cell
- Def: A binding is static if it first occurs before run time and remains unchangedthroughout program execution.
- Def: A binding is dynamic if it first occurs during execution or can change duringexecution of the program.
- Type Bindings
    - How is a type specified?
    - When does the binding take place?
    - If static, the type may be specified by either an explicit or an implicit declaration
- Def: An explicit declaration is a program statement used for declaring the types ofvariables
- Def: An implicit declaration is a default mechanism for specifying types of variables(the first appearance of the variable in the program)
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations
    - Advantage: writability
    - Disadvantage: reliability (less trouble with Perl)
- Dynamic Type Binding (JavaScript and PHP)
- Specified through an assignment statement e.g., JavaScript
- list = [2, 4.33, 6, 8];
- **list = 17.3;**
    - Advantage: flexibility (generic program units)
    - Disadvantages:
        - High cost (dynamic type checking and interpretation)
        - Type error detection by the compiler is difficult
- Type Inferencing (ML, Miranda, and Haskell)
    - Rather than by assignment statement, types are determined from the context ofthe reference
- Storage Bindings & Lifetime
    - Allocation - getting a cell from some pool of available cells
    - Deallocation - putting a cell back into the pool
- Def: The lifetime of a variable is the time during which it is bound to a particularmemory cell

- Categories of variables by lifetimes
    - Static--bound to memory cells before execution begins and remains bound tothe same memory cell throughout execution.
- e.g. all FORTRAN 77 variables, C static variables
    - Advantages: efficiency (direct addressing), history-sensitive subprogramsupport
    - Disadvantage: lack of flexibility (no recursion)
- Categories of variables by lifetimes
    - Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated.
    - If scalar, all attributes except address are statically bound
    - e.g. local variables in C subprograms and Java methods
    - Advantage: allows recursion; conserves storage
    - Disadvantages:
        - Overhead of allocation and deallocation
        - Subprograms cannot be history sensitive
        - Inefficient references (indirect addressing)
- Categories of variables by lifetimes
    - Explicit heap-dynamic--Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
    - Referenced only through pointers or references
- e.g. dynamic objects in C++ (via new and delete)all objects n Java
    - Advantage: provides for dynamic storage management
    - Disadvantage: inefficient and unreliable
- Categories of variables by lifetimes
    - Implicit heap-dynamic--Allocation and deallocation caused by assignment statements
- e.g. all variables in APL; all strings and arrays in Perl and JavaScript
    - Advantage: flexibility
    - Disadvantages:
        - Inefficient, because all attributes are dynamic
        - Loss of error detection

Type Checking

- Generalize the concept of operands and operators to include subprograms andassignments
- Type checking is the activity of ensuring that the operands of an operator are ofcompatible 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 legaltype. This automatic conversion is called a 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

Strong Typing

- Advantage of strong typing: allows the detection of the misuses of variables thatresult 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 nottype 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

Type Compatibility

- Our concern is primarily for structured types
- Def: Name type compatibility means the two variables have compatible types ifthey 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 theirtypes 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 usedifferent field names?
  - Are two array types compatible if they are the same except that the subscriptsare 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 ofthe 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
        - Derived types allow types with the same structure to be different
        - Anonymous types are all unique, even in:
- A, B : array (1..10) of INTEGER:
- **Scope**
- 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 declaredthere
- The scope rules of a language determine how references to names are associatedwith variables
- 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; thenearest 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

Static Scope Example
Static Scope

- 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 accessthem)
- 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 (temporalversus spatial)
  - References to variables are connected to declarations by searching back through
  - the chain of subprogram calls that forced execution to this pointScope Example
- MAIN
  - **declaration of xSUB1**
  - declaration of x -
- **...**
- call SUB2
- **...**

- SUB2
- **...**
  - reference to x -
- **...**

- ...
- **call SUB1**
- …
- Scope Example
- 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

- Scope and Lifetime
- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **static** variable in a C or C++ function

Referencing Environments

- Def: The referencing environment of a statement is the collection of all names thatare 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 variablesplus all visible variables in all active subprograms

- **Named Constants**
- Def: A named constant is a variable that is bound to a value only when it is boundto storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called manifestconstants) 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 statemente.g., Java
    - int sum = 0:

Expressions and Statements

Arithmetic Expressions:

- Arithmetic evaluation was one of the motivations for computers
- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.
- An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.
- In most programming languages, binary operators are **infix**, which means they appear between their operands.
- One exception is Perl, which has some operators that are **prefix**, which means they precede their operands.
- The purpose of an arithmetic expression is to specify an arithmetic computation
- An implementation of such a computation must cause two actions: fetching the operands, usually from memory, and executing arithmetic operations on those operands.
- Arithmetic expressions consist of
  - operators
  - operands
  - parentheses
  - function calls
  -

Issues for Arithmetic Expressions:

- operator precedence rules
- operator associatively rules
- order of operand evaluation
- operand evaluation side effects
- operator overloading
- mode mixing expressions

Operators:

A unary operator has one operand

- unary -, !

- A binary operator has two operands
  - +, -, *, /, %
- A ternary operator has three operands - ?:
-

Conditional Expressions:

- a ternary operator in C-based languages (e.g., C, C++)
- An example:

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

- Evaluates as if written like

if (count == 0) average = 0

else average = sum /count

Operator Precedence Rules:

- *Precedence rules* 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 Rules:

- A*ssociativity rules* define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Left to right for arithmetic operators
  - exponentiation (** or ^) 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 overriden with parentheses

Operand Evaluation Order:

1. Variables: fetch the value from memory
2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
3. Parenthesized expressions: evaluate all operands and operators first

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);
```

Functional Side Effects:

- Two possible solutions to the problem
    1. Write the language definition to disallow functional side effects
        - No two-way parameters in functions
        - No non-local references in functions
        - Advantage: it works!
        - Disadvantage: inflexibility of two-way parameters and non-local references
    2. Write the language definition to demand that operand evaluation order be fixed
        - Disadvantage: limits some compiler optimizations

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++)
    - 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)

Type Conversions

- A ***narrowing conversion*** converts an object to a type that does not include all of the values of the original type
    - e.g., float to int
- A ***widening conversion*** converts an object to a type that can include at least approximations to all of the values of the original type
    - e.g., int to float

Coercion:

- 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, widening conversions are allowed to happen implicitly
- In Ada, there are virtually no coercions in expressions

Casting:

- Explicit Type Conversions
- Called *casting* in C-based language
- Examples
  - **C:** (int) angle
  - **Ada:** Float (sum)

Note that **Ada's** syntax is similar to function calls

Errors in Expressions:

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

Relational Operators:

- Use operands of various types
- Evaluate to some Boolean representation
- Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)

Boolean Operators:

- Operands are Boolean and the result is Boolean

- **Example o**perators

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

No Boolean Type in C:

- C has no Boolean type
  - it uses int type with 0 for false and nonzero for true
- **Consequence**
  - a < b < c is a legal expression
  - the result is not what you might expect:
    - Left operator is evaluated, producing 0 or 1
    - This result is then compared with the third operand (i.e., c)

Precedence of C-based operators:

postfix ++, --

unary +, -, prefix ++, --, !

*,/,%

binary +, -

<, >, <=, >=

=, !=

&&

||

Short Circuit Evaluation:

- 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)

- Usually used for logical operators
- Problem with non-short-circuit evaluation

While (index <= length) && (LIST[index] != value)

Index++;

- When index=length, LIST [index] will cause an indexing problem (assuming LIST has length -1 elements)

Mixed-Mode Assignment:

Assignment statements can also be mixed-mode, for **example**

int a, b;

float c;

c = a / b;

- In **Java**, only widening assignment coercions are done
- In **Ada,** there is no assignment coercion

Assignment Statements:

- The general syntax

<target_var> <assign_operator> <expression>

- The assignment operator

= FORTRAN, BASIC, PL/I, C, C++, Java

:= **ALGOLs, Pascal, Ada**

- = can be bad when it is overloaded for the relational operator for equality

Compound Assignment:

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in **ALGOL;** adopted by **C**
- Example

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
  - These have side effects
- Examples

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

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

Count++ (count incremented)

-count++ (count incremented then negated - right-associative)

Assignment as an Expression:

- In C, C++, and Java, the assignment statement produces a result and can be used as operands
- An example:

While ((ch = get char ())!= EOF){…}

ch = get char() is carried out; the result (assigned to ch) is used in the condition for the while statement

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

Control Structure

- A *control structure* is a control statement and the statements whose execution

it controls

- Design question

–Should a control structure have multiple entries?

Selection Statements

- 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:

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 (continued)

- 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)

Multiple-Way Selection: Examples

- **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

elsif 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]
- Stepsize 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

Iterative Statements: 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?

Iterative Statements: 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 Selection Guarded Command: Illustrated

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

UNIT-IV

SUB PROGRAMS AND BLOCKS

Basic Definitions:

- A subprogram definition is a description of the actions of the subprogram abstraction
- A subprogram call is an explicit request that the subprogram be executed
- A subprogram header is the first line of the definition, including the name, the kind of
- subprogram, and the formal parameters
- The parameter profile of a subprogram is the number, order, and types of its parameters
- The protocol of a subprogram is its parameter profile plus, if it is a function, its return type
- 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 Param Correspondence

Two basic choices:

- Positional
- Keyword
- Sort (List => A, Length => N);
- For named association:

Advantage: order is irrelevant

Disadvantage: user must know the formal

- parameter's names

Sort (List => A, Length => N);

For named association:

Advantage: order is irrelevant

Disadvantage: user must know the formal parameter's names

Default Parameter Values

Example, in Ada:

procedure sort (list : List_Type;

length : Integer := 100);

---

sort (list => A);

Two Types of Subprograms

- Procedures provide user-defined statements, Functions provide user-defined operators

Design Issues for Subprograms

- What parameter passing methods are provided?
- Are parameter types checked?
- Are local variables static or dynamic?
- What is the referencing environment of a passed subprogram?
- Are parameter types in passed subprograms checked?
- Can subprogram definitions be nested?
- Can subprograms be overloaded?
- Are subprograms allowed to be generic?
- Is separate/independent compilation supported?
- Referencing Environments
- If local variables are stack-dynamic:
- **Advantages:**
- Support for recursion
- Storage for locals is shared among some subprograms
- **Disadvantages:**
- Allocation/deallocation time
- Indirect addressing
- Subprograms cannot be history sensitive
- Static locals are the opposite

Parameters and Parameter Passing:

Semantic Models: in mode, out mode, inout mode

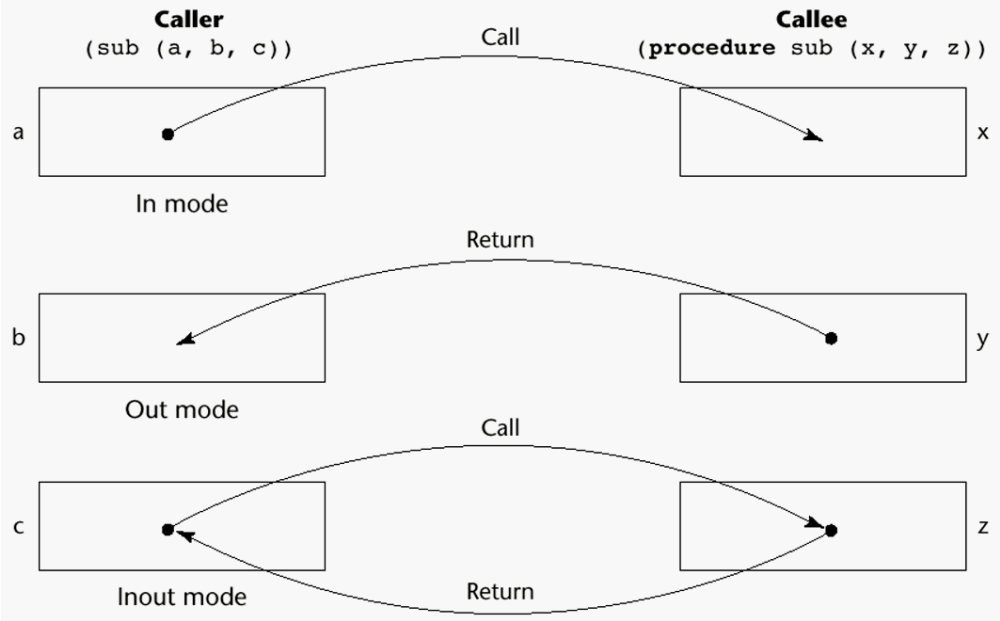
Conceptual Models of Transfer:

- Physically move a value
- Move an access path

Implementation Models:

- Pass-by-value
- Pass-by-result
- Pass-by-value-result
- Pass-by-reference
- Pass-by-name

Models of Parameter Passing



Pass-By-Value

- in mode
- Either by physical move or access path

Disadvantages of access path method: Must write-protect in the called subprogram, Accesses cost more (indirect addressing)

Disadvantages of physical move:

Requires more storage

Cost of the moves

Pass-By-Result

- out mode

Local's value is passed back to the caller

- Physical move is usually used
-

Disadvantages:

- If value is moved, time and space
- In both cases, order dependence may be a problem

```
procedure sub1(y: int, z: int);

...

sub1(x, x);
```

- Value of x in the caller depends on order of assignments at the return

Pass-By-Value-Result

inout mode

Physical move, both ways Also called pass-by-copy

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

Advantage: passing process is efficient

Disadvantages:

Slower accesses can allow aliasing: Actual parameter collisions: sub1(x, x); Array element collisions: sub1(a[i], a[j]); /* if i = j */ Collision between formals and globals Root cause of all of these is: The called subprogram is provided wider access to non locals than is necessary

Pass-by-value-result does not allow these aliases (but has other problems!)

Pass-By-Name multiple modes 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

Purpose: flexibility of late binding

Resulting semantics:

If actual is a scalar variable, it is pass-by-reference

If actual is a constant expression, it is pass-by-value

If actual is an array element, it is like nothing else

If actual is an expression with a reference to a variable that is also accessible in the program, it is also like nothing else

Pass-By-Name Example 1

```
procedure sub1(x: int; y: int);

begin

x := 1;

y := 2;

x := 2;

y := 3;

end;

sub1(i, a[i]);
```

Pass-By-Name Example 2

- Assume k is a global variable

```
procedure sub1(x: int; y: int; z: int);

begin

k := 1;

y := x;
```

```
k := 5;

z := x;

end;

sub1(k+1, j, i);
```

Disadvantages of Pass-By-Name

- Very inefficient references
- Too tricky; hard to read and understand

Param Passing: Language Examples

FORTRAN, Before 77, pass-by-reference 77—scalar variables are often passed by value result

ALGOL 60 Pass-by-name is default; pass-by-value is optional

ALGOL W: Pass-by-value-result ,C: Pass-by-value Pascal and Modula-2: Default is pass-by value;,pass-by-reference is optional

Param Passing: PL Example

C++: Like C, but also allows reference type parameters, which provide the efficiency of pass-by-reference with in-mode semantics

Ada

All three semantic modes are available If out, it cannot be referenced If in, it cannot be assigned Java Like C++, except only references

Type Checking Parameters

Now considered very important for reliability

FORTRAN 77 and original C: none

Pascal, Modula-2, FORTRAN 90, Java, and Ada: it is always required

ANSI C and C++: choice is made by the user

Implementing Parameter Passing

ALGOL 60 and most of its descendants use the runtime stack Value—copy it to the stack; references are indirect to the stack Result—sum, Reference—regardless of form, put the address in the stack

Name:

Run-time resident code segments or subprograms evaluate the address of the parameter Called for each reference to the formal, these are called thunks Very expensive, compared to reference or value-result

Ada Param Passing Implementations

Simple variables are passed by copy (valueresult) Structured types can be either by copy or reference This can be a problem, because Aliasing differences (reference allows aliases, but value-result does not) Procedure termination by error can produce different actual parameter results Programs with such errors are “erroneous”

Multidimensional Arrays as Params

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

C and C++

Programmer is required to include the declared sizes of all but the first subscript in the actual parameter , This 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, which is in terms of the size

More Array Passing Designs

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 Pre-90 FORTRAN , Formal parameter declarations for arrays can include passed parameters

```
SUBPROGRAM SUB(MATRIX, ROWS, COLS, RESULT)
  INTEGER ROWS, COLS
  REAL MATRIX (ROWS, COLS), RESULT
```

...

END

Design Considerations for Parameter Passing

- Efficiency
- One-way or two-way
- These two are in conflict with one another!
- Good programming => limited access to
- variables, which means one-way whenever possible
- Efficiency => pass by reference is fastest way to pass structures of significant size Also, functions should not allow reference Parameters

Subprograms As Parameters: Issues

Are parameter types checked?

Early Pascal and FORTRAN 77 do not Later versions of Pascal, Modula-2, and FORTRAN 90 do Ada does not allow subprogram parameters C and C++ - pass pointers to functions; parameters can be type checked

What is the correct referencing?

environment for a subprogram that was sent as a parameter?

Possibilities:

It is that of the subprogram that called it (shallow binding)

It is that of the subprogram that declared it (deep binding)

It is that of the subprogram that passed it (ad hocbinding, never been used)

For static-scoped languages, deep binding is most natural

For dynamic-scoped languages, shallow binding is most natural

Overloading

An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment C++ and Ada have overloaded subprograms built-in, and users can write their own overloaded subprograms

Generic Subprograms

1. A generic or polymorphic subprogram is one that takes parameters of different types on different activations Overloaded subprograms provide ad hoc polymorphism
2. 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
3. See Ada generic and C++ template examples in text
4. Independent compilation is compilation of some of the units of a program separately from the rest of the program, without the benefit of interface information
5. Separate compilation is compilation of some of the units of a program separately from the rest of the program, using interface information to check the correctness of the interface between the two parts

Language Examples:

- FORTRAN II to FORTRAN 77: independent
- FORTRAN 90, Ada, Modula-2, C++: separate
- Pascal: allows neither

Functions

Design Issues:

- Are side effects allowed?
- Two-way parameters (Ada does not allow)
- Nonlocal reference (all allow)
- What types of return values are allowed?

- FORTRAN, Pascal, Modula-2: only simple types
- **C**: any type except functions and arrays
- **Ada:** any type (but subprograms are not types)
- **C++ and Java**: like C, but also allow classes to be r**eturned**

Accessing Nonlocal Environments

The nonlocal variables of a subprogram are those that are visible but not declared in the subprogram

Global variables are those that may be visible in all of the subprograms of a program

Methods for Accessing Non locals

FORTRAN COMMON

The only way in pre-90 FORTRANs to access nonlocal variables

Can be used to share data or share storage

Static scoping

External declarations: C

- Subprograms are not nested
- Globals are created by external declarations (they are simply defined outside any function)
- Access is by either implicit or explicit declaration
- Declarations (not definitions) give types to externally defined variables (and say they are defined elsewhere)
- External modules: Ada and Modula-2:
- Dynamic Scope:

User-Defined Overloaded Operators

Nearly all programming languages have overloaded operators

Users can further overload operators in C++ and Ada not carried over into Java)

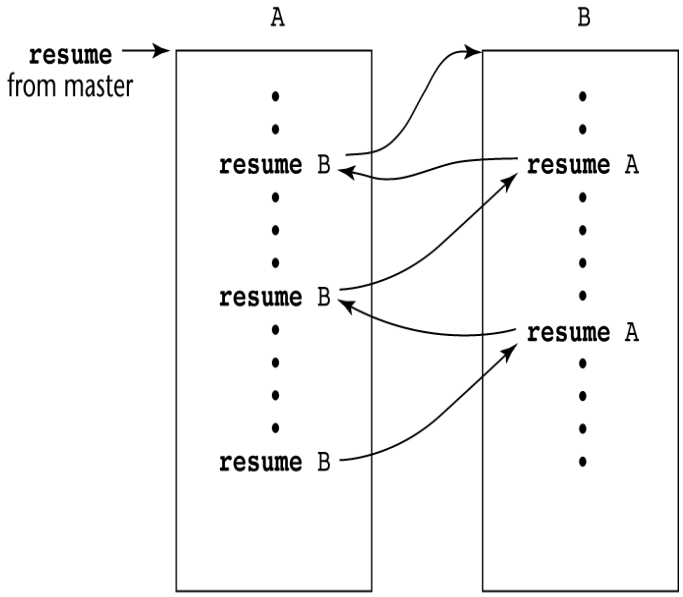
Ada Example (where Vector_Type is an array of Integers):

```
function "*"(a, b : inVector_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 "*";
```

Are user-defined overloaded operators good or bad?

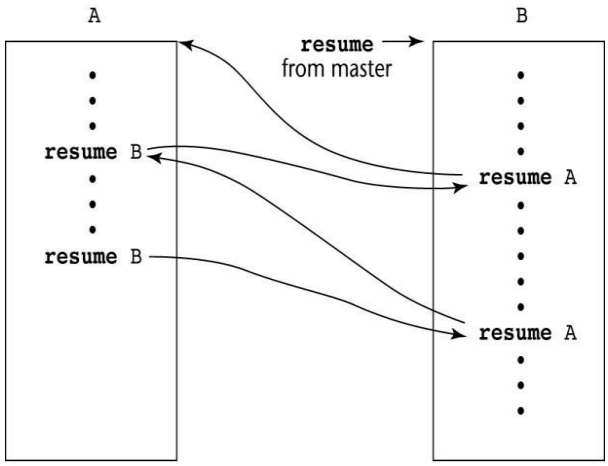
Coroutines:

Coroutine is a subprogram that has multiple entries and controls them itself Also called symmetric control 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. Typically, coroutines repeatedly resume each other, possibly forever. Coroutines provide quasi concurrent execution of program units (the coroutines) Their execution is interleaved, but not overlapped


A
B
resume
from master
resume B
resume B
resume B
resume A
resume A

(a)

Coroutines Illustrated: Possible Execution Controls



Coroutines Illustrated: Possible Execution Controls with Loops

UNIT-V

ABSTRACT DATA TYPES

The Concept of Abstraction

• 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*

Introduction to Data Abstraction

• 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?

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
```

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

Language Examples: C++ (continued)

• Information Hiding

— *Private* clause for hidden entities

— *Public* clause for interface entities

— *Protected* clause for inheritance (Chapter 12)

Language Examples: C++ (continued)

• 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

Language Examples: C++ (continued)

• 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

Language Examples: C++ (continued)

• 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

Language Examples: C# (continued)

• 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 property
                get {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;
```

Parameterized Abstract Data Types

• 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);
```

Parameterized ADTs in C++ (continued)

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

```
template <class Type>
class stack {
  private:
    Type *stackPtr;
    const int maxLen;
    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

- 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

Object-Oriented Programming

- Abstract data types
- Inheritance
- —Inheritance is the central theme in OOP and languages that support it
- Polymorphism

Inheritance

- Productivity increases can come from reuse
  - —ADTs are difficult to reuse—always need changes
  - —All ADTs are independent and at the same level
- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
- Inheritance addresses both of the above concerns--reuse ADTs after minor changes and define classes in a hierarchy

Object-Oriented Concepts

- ADTs are usually called *classes*
- Class instances are called *objects*
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *superclass*
- Subprograms that define operations on objects are called *methods*

Object-Oriented Concepts (continued)

- Calls to methods are called *messages*
- The entire collection of methods of an object is called its *message protocol* or *message interface*
- Messages have two parts--a method name and the destination object
- In the simplest case, a class inherits all of the entities of its parent

Object-Oriented Concepts (continued)

- Inheritance can be complicated by access controls to encapsulated entities
  - —A class can hide entities from its subclasses
  - —A class can hide entities from its clients
  - —A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method

— The new one *overrides* the inherited oneThe

— method in the parent is *overriden*

Object-Oriented Concepts (continued)

- There are two kinds of variables in a class:

— *Class variables* - one/class

— *Instance variables* - one/object

- There are two kinds of methods in a class:

— *Class methods* – accept messages to the class

— *Instance methods* – accept messages to objects

- Single vs. Multiple Inheritance

- One disadvantage of inheritance for reuse:

— Creates interdependencies among classes that complicate maintenance

Dynamic Binding

- A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants

- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic

- Allows software systems to be more easily extended during both development and maintenance

Dynamic Binding Concepts

- An *abstract method* is one that does not include a definition (it only defines a protocol)

- An *abstract class* is one that includes at least one virtual method

- An abstract class cannot be instantiated

Design Issues for OOP Languages

- The Exclusivity of Objects

- Are Subclasses Subtypes

- Type Checking and Polymorphism

- Single and Multiple Inheritance

• Object Allocation and DeAllocation

• Dynamic and Static Binding

• Nested Classes

The Exclusivity of Objects

• Everything is an object

—Advantage - elegance and purity

—Disadvantage - slow operations on simple objects

• Add objects to a complete typing system

—Advantage - fast operations on simple objects

—Disadvantage - results in a confusing type system (two kinds of entities)

• Include an imperative-style typing system for primitives but make everything else objects

—Advantage - fast operations on simple objects and a relatively small typing system

—Disadvantage - still some confusion because of the two type systems

Are Subclasses Subtypes?

• Does an —is-a|| relationship hold between a parent class object and an object of the subclass?

—If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object

• A derived class is a subtype if it has an is-a relationship with its parent class

—Subclass can only add variables and methods and override inherited methods in —compatible|| ways

Type Checking and Polymorphism

• Polymorphism may require dynamic type checking of parameters and the return value

—Dynamic type checking is costly and delays error detection

• If overriding methods are restricted to having the same parameter types and

return type, the checking can be static

Single and Multiple Inheritance

- Multiple inheritance allows a new class to inherit from two or more classes
- Disadvantages of multiple inheritance:

—Language and implementation complexity (in part due to name collisions)

—Potential inefficiency - dynamic binding costs more with multiple inheritance (but not much)

- **Advantage:**

—Sometimes it is quite convenient and valuable

Allocation and DeAllocation of Objects

- From where are objects allocated?

—If they behave line the ADTs, they can be allocated from anywhere

- Allocated from the run-time stack
- Explicitly create on the heap (via new)

—If they are all heap-dynamic, references can be uniform thru a pointer or reference variable

- Simplifies assignment - dereferencing can be implicit

—If objects are stack dynamic, there is a problem with regard to subtypes

- Is deallocation explicit or implicit?

Dynamic and Static Binding

- Should all binding of messages to methods be dynamic?

—If none are, you lose the advantages of dynamic binding

—If all are, it is inefficient

- Allow the user to specify

Nested Classes

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes

—Can the new class be nested inside the class that uses it?

—In some cases, the new class is nested inside a subprogram rather than directly in another class

• Other issues:

—Which facilities of the nesting class should be visible to the nested class and vice versa

Support for OOP in Smalltalk

• Smalltalk is a pure OOP language

—Everything is an object

—All objects have local memory

—All computation is through objects sending messages to objects

—None of the appearances of imperative languages

—All objected are allocated from the heap

—All deallocation is implicit

Support for OOP in Smalltalk (continued)

• Type Checking and Polymorphism

—All binding of messages to methods is dynamic

• The process is to search the object to which the message is sent for the method; if not found, search the superclass, etc. up to the system class which has no superclass

—The only type checking in Smalltalk is dynamic and the only type error occurs when a message is sent to an object that has no matching method

Support for OOP in Smalltalk (continued)

• Inheritance

—A Smalltalk subclass inherits all of the instance variables, instance methods, and class methods of its superclass

—All subclasses are subtypes (nothing can be hidden)

—All inheritance is implementation inheritance

— No multiple inheritance

Support for OOP in Smalltalk (continued)

- Evaluation of Smalltalk

— The syntax of the language is simple and regular

— Good example of power provided by a small language

— Slow compared with conventional compiled imperative languages

— Dynamic binding allows type errors to go undetected until run time

— Introduced the graphical user interface

— Greatest impact: advancement of OOP

Support for OOP in C++

- General Characteristics:

— Evolved from C and SIMULA 67

— Among the most widely used OOP languages

— Mixed typing system

— Constructors and destructors

— Elaborate access controls to class entities

Support for OOP in C++ (continued)

- Inheritance

— A class need not be the subclass of any class

— Access controls for members are

— Private (visible only in the class and friends) (disallows subclasses from being subtypes)

— Public (visible in subclasses and clients)

— Protected (visible in the class and in subclasses, but not clients)

Support for OOP in C++ (continued)

- In addition, the subclassing process can be declared with access controls

(private or public), which define potential changes in access by subclasses

— Private derivation - inherited public and protected members are private in the subclasses

— Public derivation public and protected members are also public and protected in subclasses

Inheritance Example in C++

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class { ... };
//      In this one, b and y are protected and
//      c and z are public

class subclass_2 : private base_class { ... };
//     In this one, b, y, c, and z are private,
//     and no derived class has access to any
//     member of base_class
```

Reexportation in C++

• A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (**::**), e.g.,

```
class subclass_3 : private base_class {
        base_class :: c;
                ...
}
```

Reexportation (continued)

• One motivation for using private derivation

— A class provides members that must be visible, so they are defined to be public members; a derived class adds some new members, but does not want its clients to see the members of the parent class, even though they had to be public in the parent class definition

Support for OOP in C++ (continued)

• Multiple inheritance is supported

—If there are two inherited members with the same name, they can both be referenced using the scope resolution operator

Support for OOP in C++ (continued)

• Dynamic Binding

—A method can be defined to be virtual, which means that they can be called through polymorphic variables and dynamically bound to messages

—A pure virtual function has no definition at all

—A class that has at least one pure virtual function is an *abstract class*

Support for OOP in C++ (continued)

• Evaluation

—C++ provides extensive access controls (unlike Smalltalk)

—C++ provides multiple inheritance

—In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound

• Static binding is faster!

—Smalltalk type checking is dynamic (flexible, but somewhat unsafe)

—Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++

Support for OOP in Java

• Because of its close relationship to C++, focus is on the differences from that language

• General Characteristics

—All data are objects except the primitive types

—All primitive types have wrapper classes that store one data value

—All objects are heap-dynamic, are referenced through reference variables, and most are allocated with new

—A finalize method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object

Support for OOP in Java (continued)

- Inheritance

—Single inheritance supported only, but there is an abstract class category that provides some of the benefits of multiple inheritance (interface)

—An interface can include only method declarations and named constants, e.g.,

```
public interface Comparable {
        public int comparedTo (Object b);
}
```

—Methods can be **final** (cannot be overriden)

Support for OOP in Java (continued)

- Dynamic Binding

—In Java, all messages are dynamically bound to methods, unless the method is final (i.e., it cannot be overriden, therefore dynamic binding serves no purpose)

—Static binding is also used if the methods is static or private both of which disallow overriding

Support for OOP in Java (continued)

- Several varieties of nested classes
- All are hidden from all classes in their package, except for the nesting class
- Nested classes can be anonymous
- A local nested class is defined in a method of its nesting class

—No access specifier is used

Support for OOP in Java (continued)

- Evaluation

—Design decisions to support OOP are similar to C++

—No support for procedural programming

—No parentless classes

—Dynamic binding is used as —normal‖ way to bind method calls to method definitions

—Uses interfaces to provide a simple form of support for multiple inheritance

Support for OOP in C#

• General characteristics

—Support for OOP similar to Java

—Includes both classes and structs

—Classes are similar to Java's classes

—structs are less powerful stack-dynamic constructs (e.g., no inheritance)

Support for OOP in C# (continued)

• Inheritance

—Uses the syntax of C++ for defining classes

—A method inherited from parent class can be replaced in the derived class by marking its definition with new

—The parent class version can still be called explicitly with the prefix base: base.Draw()

Support for OOP in C#

• Dynamic binding

—To allow dynamic binding of method calls to methods:

• The base class method is marked virtual

• The corresponding methods in derived classes are marked override

—Abstract methods are marked abstract and must be implemented in all subclasses

—All C# classes are ultimately derived from a single root class, Object

Support for OOP in C# (continued)

• Nested Classes

—A C# class that is directly nested in a nesting class behaves like a Java static nested class

—C# does not support nested classes that behave like the non-static classes of

Java

Support for OOP in C#

• Evaluation

—C# is the most recently designed C-based OO language

—The differences between C#'s and Java's support for OOP are relatively minor

Support for OOP in Ada 95

• General Characteristics

—OOP was one of the most important extensions to Ada 83

—Encapsulation container is a package that defines a *tagged type*

—A tagged type is one in which every object includes a tag to indicate during execution its type (the tags are internal)

—Tagged types can be either private types or records

—No constructors or destructors are implicitly called

Support for OOP in Ada 95 (continued)

• Inheritance

—Subclasses can be derived from tagged types

—New entities are added to the inherited entities by placing them in a record definition

—All subclasses are subtypes

—No support for multiple inheritance

• A comparable effect can be achieved using generic classes

Example of a Tagged Type

```
Package Person_Pkg is
  type Person is tagged private;
  procedure Display(P : in out Person);
  private
    type Person is tagged
      record
        Name : String(1..30);
        Address : String(1..30);
        Age : Integer;
      end record;
```

```
end Person_Pkg;
with Person_Pkg; use Person_Pkg;
 package Student_Pkg is
   type Student is new Person with
     record
       Grade_Point_Average : Float;
       Grade_Level : Integer;
     end record;
   procedure Display (St: in Student);
 end Student_Pkg;

// Note: Display is being overridden from Person_Pkg
```

Support for OOP in Ada 95 (continued)

- Dynamic Binding

—Dynamic binding is done using polymorphic variables called classwide types

- For the tagged type **Prtdon**, the classwide type is **Person‘ class**

—Other bindings are static

—Any method may be dynamically bound

—Purely abstract base types can be defined in Ada 95 by including the reserved word abstract

Support for OOP in Ada 95 (continued)

- Evaluation

—Ada offers complete support for OOP

—C++ offers better form of inheritance than Ada

—Ada includes no initialization of objects (e.g., constructors)

—Dynamic binding in C-based OOP languages is restricted to pointers and/or references to objects; Ada has no such restriction and is thus more orthogonal

Implementing OO Constructs

- Two interesting and challenging parts

—Storage structures for instance variables

—Dynamic binding of messages to methods

Instance Data Storage

• Class instance records (CIRs) store the state of an object

—Static (built at compile time)

• If a class has a parent, the subclass instance variables are added to the parent CIR

• Because CIR is static, access to all instance variables is done as it is in records

—Efficient

Dynamic Binding of Methods Calls

• Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR

—Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR

—The storage structure is sometimes called *virtual method tables* (vtable)

—Method calls can be represented as offsets from the beginning of the vtable

Summary

• OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding

• Major design issues: exclusivity of objects, subclasses and subtypes, type checking and polymorphism, single and multiple inheritance, dynamic binding, explicit and implicit de-allocation of objects, and nested classes

• Smalltalk is a pure OOL

• C++ has two distinct type system (hybrid)

• Java is not a hybrid language like C++; it supports only OO programming

• C# is based on C++ and Java

• Implementing OOP involves some new data structures

Exception Handling

Introduction to Exception 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

Basic Concepts

- 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 are exception handlers specified and what is their scope?

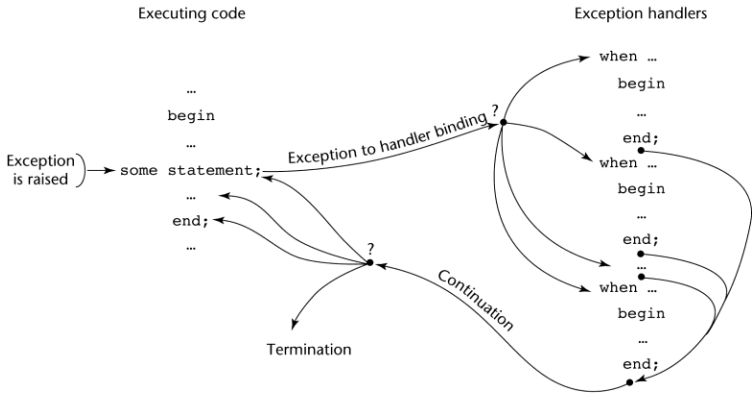
• How is an exception occurrence bound to an exception handler?

• 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?

Exception Handling Control Flow



Exception Handling in Ada

• 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++

Exception Handling in C++

- 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

Exception Handling in Java

• 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
    - Thrown by the Java interpreter for events such as heap overflow
    - Never handled by user programs
  - Exception
    - User-defined exceptions are usually subclasses of this
    - 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

UNIT-VI

FUNCTIONAL PROGRAMMING LANGUAGES

Functional Programming Language Introduction

• 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

λ(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., (λ(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 ≡ f ° g

which means h (x) ≡ f (g (x))

For f (x) ≡ x + 2 and g (x) ≡ 3 * x,

h ≡ f ° 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: α

For h (x) $\equiv$ x * x

α(h, (2, 3, 4)) yields (4, 9, 16)

Fundamentals of Functional Programming Languages

• 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

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

ML

- 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

Haskell

• 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 [1..n]
```

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

```
numbermember [] b =
False
member(a:x) b=(a ==
b)||member x bsquares = [n *
n | n ← [0..]]
member squares 16
```

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 willalways work:

```
member2 (m:x) n
        | m < n = member2 x n
        | m == n = True
        | otherwise = False
```

Applications of Functional Languages

- 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

Logic Programming Languages

Introduction

• *Logic* programming languages, sometimes called *declarative* programminglanguages

• Express programs in a form of symbolic logic

• Use a logical inferencing process to produce results

• *Declarative* rather that *procedural*:

— Only specification of *results* are stated (not detailed *procedures* for producingthem)

Proposition

• A logical statement that may or may not be true

—Consists of objects and relationships of objects to each other

Symbolic Logic

• Logic which can be used for the basic needs of formal logic:

—Express propositions

—Express relationships between propositions

—Describe how new propositions can be inferred from other propositions

• Particular form of symbolic logic used for logic programming called *predicatecalculus*

Object Representation

• Objects in propositions are represented by simple terms: either constants orvariables

• *Constant*: a symbol that represents an object

• *Variable*: a symbol that can represent different objects at different times

—Different from variables in imperative languages

Compound Terms

• *Atomic propositions* consist of compound terms

• *Compound term*: one element of a mathematical relation, written like a mathematical function

- Mathematical function is a mapping

- Can be written as a table

Parts of a Compound Term

- Compound term composed of two parts

  - Functor: function symbol that names the relationship

  - Ordered list of parameters (tuple)

- Examples:
  student(jon)
  like(seth, OSX)
  like(nick, windows)
  like(jim, linux)

Forms of a Proposition

- Propositions can be stated in two forms:

  - *Fact*: proposition is assumed to be true

  - *Query*: truth of proposition is to be determined

- Compound proposition:

  - Have two or more atomic propositions

  - Propositions are connected by operators

Logical Operators

Name	Symbol	Example	Meaning
negation	$\neg$	$\neg a$	not a
conjunction	$\cap$	$a \cap b$	a and b
disjunction	$\cup$	$a \cup b$	a or b
equivalence	$\equiv$	$a \equiv b$	a is equivalent to b
implication	$\supset$ $\subset$	$a \supset b$ $a \subset b$	a implies b b implies a

Quantifiers

Name	Example	Meaning
universal ∀	X.P	For all X, P is true
Existential ∃	X.P	There exists a value of X such that P is true

Clausal Form

• Too many ways to state the same thing

• Use a standard form for propositions

• *Clausal form*:

$$-B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m|$$

— means if all the As are true, then at least one B is true

• *Antecedent*: right side

• *Consequent*: left side

Predicate Calculus and Proving Theorems

• A use of propositions is to discover new theorems that can be inferred fromknown axioms and theorems

• *Resolution*: an inference principle that allows inferred propositions to be computed from given propositions

Resolution

• *Unification*: finding values for variables in propositions that allows matchingprocess to succeed

• *Instantiation*: assigning temporary values to variables to allow unification tosucceed

• After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

Theorem Proving

• Basis for logic programming

• When propositions used for resolution, only restricted form can be used

• *Horn clause* - can have only two forms

- *Headed*: single atomic proposition on left side
- *Headless*: empty left side (used to state facts)

• Most propositions can be stated as Horn clauses

Overview of Logic Programming

• Declarative semantics

- There is a simple way to determine the meaning of each statement
- Simpler than the semantics of imperative languages

• Programming is nonprocedural

- Programs do not state now a result is to be computed, but rather theform of the result

The Origins of Prolog

• University of Aix-Marseille

- Natural language processing

• University of Edinburgh

- Automated theorem proving

Terms

• Edinburgh Syntax

• *Term*: a constant, variable, or structure

• *Constant*: an atom or an integer

• *Atom*: symbolic value of Prolog

• Atom consists of either:

- a string of letters, digits, and underscores beginning with a lowercase letter
- a string of printable ASCII characters delimited by apostrophes

Terms: Variables and Structures

• *Variable*: any string of letters, digits, and underscores beginning with anuppercase letter

• *Instantiation*: binding of a variable to a value

—Lasts only as long as it takes to satisfy one complete goal

• *Structure*: represents atomic propositionfunctor(*parameter list*)

Fact Statements

• Used for the hypotheses

• Headless Horn clauses
female(shelley).
male(bill).
father(bill, jake).

Rule Statements

• Used for the hypotheses

• Headed Horn clause

• Right side: *antecedent* (***if*** part)

—May be single term or conjunction

• Left side: *consequent* (***then*** part)

—Must be single term

• *Conjunction*: multiple terms separated by logical AND operations

(implied)

Example Rules

ancestor(mary,shelley):- mother(mary,shelley).

• Can use variables (*universal objects*) to generalize meaning:parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).
grandparent(X,Z):- parent(X,Y), parent(Y,Z).
sibling(X,Y):- mother(M,X), mother(M,Y),father(F,X), father(F,Y).

Goal Statements

• For theorem proving, theorem is in form of proposition that we want systemto prove or disprove – *goal statement*

• Same format as headless Horn
man(fred)

- Conjunctive propositions and propositions with variables also legal goalsfather(X,mike)

Inferencing Process of Prolog

- Queries are called goals
- If a goal is a compound proposition, each of the facts is a subgoal
- To prove a goal is true, must find a chain of inference rules and/or facts.

For goal Q:

```
B :- A
C :- B
...
Q :- P
```

- Process of proving a subgoal called matching, satisfying, or resolution

Approaches

- *Bottom-up resolution, forward chaining*
  - Begin with facts and rules of database and attempt to find sequencethat leads to goal
  - Works well with a large set of possibly correct answers
- *Top-down resolution, backward chaining*
  - Begin with goal and attempt to find sequence that leads to set of factsin database
  - Works well with a small set of possibly correct answers
- Prolog implementations use backward chaining

Subgoal Strategies

- When goal has more than one subgoal, can use either
  - Depth-first search: find a complete proof for the first subgoal beforeworking on others
  - Breadth-first search: work on all subgoals in parallel
- Prolog uses depth-first search
  - Can be done with fewer computer resources

Backtracking

• With a goal with multiple subgoals, if fail to show truth of one of subgoals,reconsider previous subgoal to find an alternative solution: *backtracking*

• Begin search where previous search left off

• Can take lots of time and space because may find all possible proofs to everysubgoal

Simple Arithmetic

• Prolog supports integer variables and integer arithmetic

• is operator: takes an arithmetic expression as right operand and variable as
left operand

A is B / 17 + C

• Not the same as an assignment statement!

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-    speed(X,Speed),time(X,Time),
                        Y is Speed * Time.
```

Trace

• Built-in structure that displays instantiations at each step

• *Tracing model* of execution - four events:

— *Call* (beginning of attempt to satisfy goal)

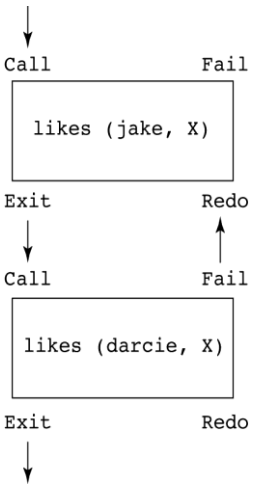
— *Exit* (when a goal has been satisfied)

— *Redo* (when backtrack occurs)

— *Fail* (when goal fails)

Example

likes(jake,chocolate).
likes(jake,apricots).
likes(darcie,licorice).
likes(darcie,apricots).
trace.
likes(jake,X),
likes(darcie,X).



List Structures

• Other basic data structure (besides atomic propositions we have alreadyseen): list

• *List* is a sequence of any number of elements

• Elements can be atoms, atomic propositions, or other terms (including other lists)

[apple, prune, grape,kumquat]

[] *(empty list)*

[X | Y] *(head X and tail Y)*

Append Example

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
                                    append (List_1, List_2, List_3).
```

Reverse Example

```
reverse([], []).
reverse([Head | Tail], List) :-reverse (Tail, Result),

append (Result, [Head], List).
```

Deficiencies of Prolog

• Resolution order control

• The closed-world assumption

• The negation problem

• Intrinsic limitations

Applications of Logic Programming

• Relational database management systems

• Expert systems

• Natural language processing