PRINCIPLES OF PROGRAMMING LANGUAGES

Prepared By:

Dr. Shanmugasundaram

Assistant Professor

CS Department

Jazan University

❖ *Reasons for Studying Concepts of Programming Languages*

- Increased ability to express ideas.
- Improved background for choosing appropriate languages.
- Increased ability to learn new languages.
- Better understanding of significance of implementation.
- Better use of languages that are already known.
- Overall advancement of computing.

❖ *Programming Domains*

- ***Scientific Applications***
  - Large numbers of floating point computations; use of arrays.
  - Example:Fortran.
- ***Business Applications***
  - Produce reports, use decimal numbers and characters.
  - Example:COBOL.
- ***Artificial intelligence***
  - Symbols rather than numbers manipulated; use of linked lists.
  - Example:LISP.

❖ *Programming Domains*

- ***System programming***

  - Need effieciency because of continous use.

  - Example:C

- ***Web Software***

  -Eclectic collection of languages: markup(example:XHTML),scripting(example:PHP), general-purpose(example:JAVA).

❖ Language Evaluation Criteria

- ***Readability***:

➢ The ease with which programs can be read and understood.

- ***Writability***:

➢ The ease with which a language can be used to create programs.

- ***Reliability***:

➢ Conformance to specifications (i.e., performs to its specifications).

- **Cost**:

➢ The ultimate total cost.

❖ Evaluation Criteria: Readability

➔ ***Overall simplicity***
- ◆ *A manageable set of features and constructs.*
- ◆ *Minimal feature multiplicity .*
- ◆ *Minimal operator overloading.*

➔ ***Orthogonality***
- ◆ *A relatively small set of primitive constructs can be combined in a relatively small number of ways*
- ◆ *Every possible combination is legal*

➔ ***Data types***
- ◆ *Adequate predefined data types.*

❖ Evaluation Criteria:Readability

➔ Syntax considerations

-Identifier forms:flexible composition.

-Special words and methods of forming compound statements.

-Form and meaning:self-descriptive constructs,meaningful keywords.

❖ Evaluation Criteria: Writability

- **Simplicity and orthogonality**
  - Few constructs, a small number of primitives, a small set of rules for combining them.
- **Support for abstraction**

-The ability to define and use complex structures or operations in ways that allow details to be ignored.

- **Expressivity**
  - A set of relatively convenient ways of specifying operations.
  - Strength and number of operators and predefined functions.

❖ *Evaluation Criteria: Reliability*

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

❖ *Evaluation Criteria: Cost*

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

❖ *Language Categories*

- **Imperative**
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- **Functional**
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- **Logic**
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- **Markup/programming hybrid**
  - Markup languages extended to support some programming
  - Examples: JSTL, XSLT

❖ *Language Design Trade-Offs*

- Reliability vs. cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
- Readability vs. writability

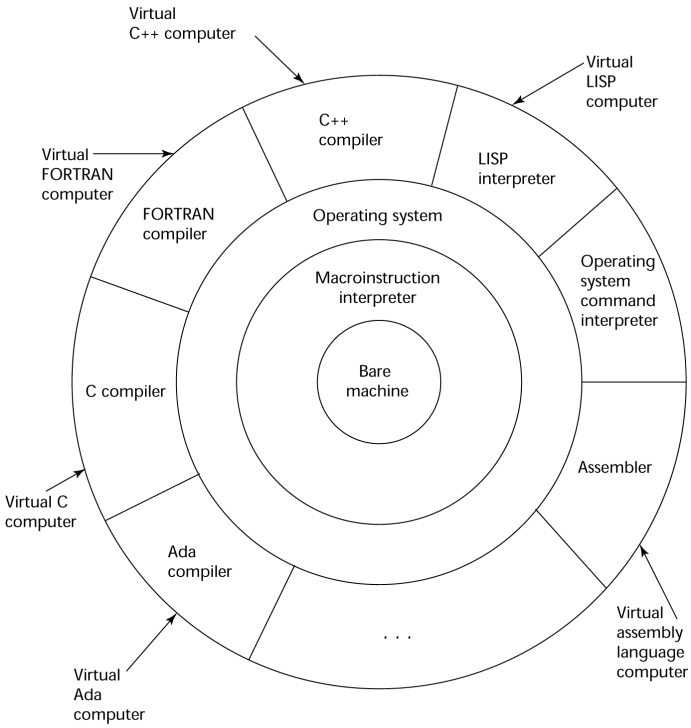
  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Example: C++ pointers are powerful and very flexible but are unreliable

❖ 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

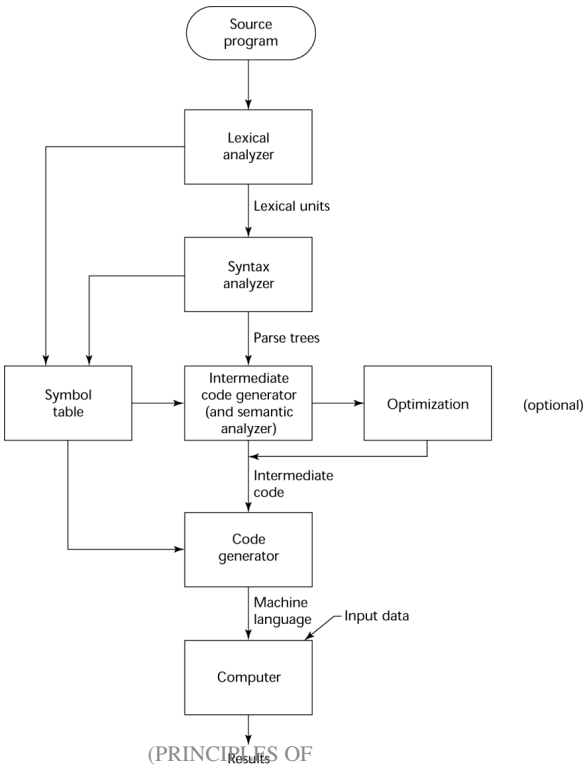
The operating system and language implementation are layered over machine interface of a computer



Compilation

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

The Compilation Process



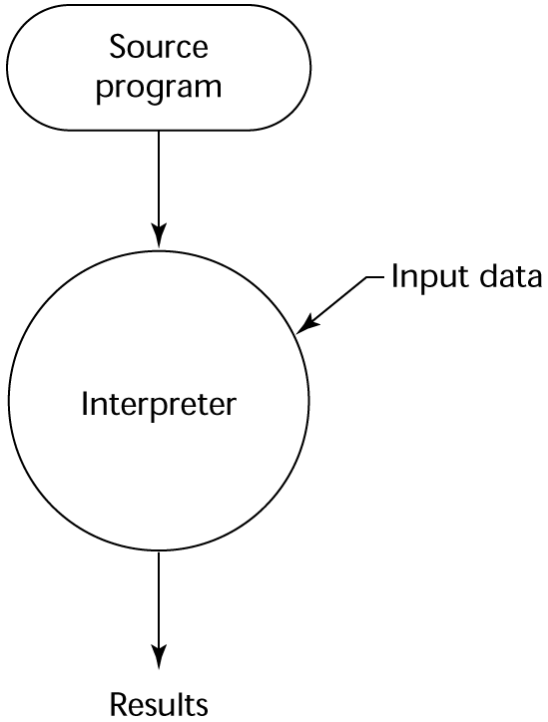
Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

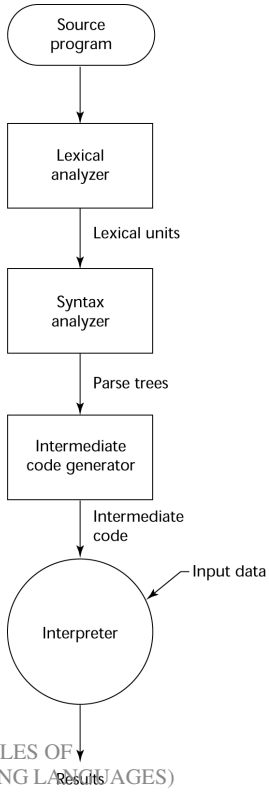
Pure Interpretation Process



Hybrid Implementation Systems

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

Hybrid Implementation Process



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- 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

Programming Environments

- A collection of tools used in software development
- UNIX
  - An older operating system and tool collection
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX
- Microsoft Visual Studio.NET
  - A large, complex visual environment
- Used to build Web applications and non-Web applications in any .NET language
- NetBeans
  - Related to Visual Studio .NET, except for Web applications in Java

CHAPTER 2 : SYNTAX AND SEMANTIC

CONTENTS

Topics:

Introduction

Syntax:

The form or structure of the expressions, statements, and program units

Semantics:

The meaning of the expressions, statements, and program units.

Syntax and semantics provide a language's definition

Users of a language definition

- Other language designers
- Implementers
- Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- A sentence is a string of characters over some alphabet
- A language is a set of sentences
- A lexeme is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A token is a category of lexemes (e.g., identifier)

Formal Definition of Languages

Recognizers

- A recognition device reads input strings of the language and decides whether the input strings belong to the language

Generators

- A device that generates sentences of a language
- One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

Formal Methods of Describing Syntax

Backus-Naur Form and Context-Free Grammars

- Developed by Noam Chomsky in the mid-1950s.

A context Free Grammar (CFG) is a 4-tuple such that- **G = (V , T , P , S)**
where-
V = Finite non-empty set of variables / non-terminal symbols
T = Finite set of terminal symbols
P = Finite non-empty set of production rules of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$
S = Start symbol

•A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
•A set of tokens, known as **terminal symbols** (T). Terminals are the basic symbols from which strings are formed.
•A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.

Example Context free grammar

Consider a grammar G = (V , T , P , S) where-

•V = { S }
•T = { a , b }
•P = { $S \rightarrow aSbS$, $S \rightarrow bSaS$, $S \rightarrow \in$ }
•S = { S }

So, Language of this grammar is-

L(G) = { ∈ , ab , ba , aabb , bbaa , abab , baba , ... }

This grammar is an example of a context free grammar.

•It generates the strings having equal number of a's and b's.
•In the above grammar The Nonterminal is : S
•Terminal is : a,b, ∈
•Start Symbol is S
•And the Number of production is 3

Application of CFG (Context free grammar)

Context Free Grammar (CFG) is of great practical importance. It is used for following purposes-

- For defining programming languages
- For parsing the program by constructing syntax tree
- For translation of programming languages
- For describing arithmetic expressions
- For construction of compilers.

Backus-Naur Form (BNF)

BNF (Backus–Naur Form) is a context-free grammar commonly used by developers of programming languages to specify the syntax rules of a language.

John Backus was a program language designer who devised a notation to document IAL (an early implementation of Algol).

Peter Naur later worked on Backus' findings, and the notation was jointly credited to both computer scientists.

BNF uses a range of symbols and expressions to create **production rules**.

A simple BNF production rule might look like this:

<digit> ::= 0|1|2|3|4|5|6|7|8|9

This would be interpreted as: *a digit can be defined as 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9*

The chevrons (< >) are used to denote a **non-terminal symbol**. If a non-terminal symbol appears on the right side of the production rules, it means that there will be another production rule (or set of rules) to define its replacement.

There is the production for any grammar as follows:

1. S → aSa
2. S → bSb
3. S → c

In BNF, we can represent above grammar as follows:

1. S → aSa| bSb| c

BNF Fundamentals

In the above grammar

Non-terminals: BNF abstractions = S

Terminals: lexemes and tokens = a,b,c

Grammar: a collection of rules

Examples of BNF rules:

<ident_list> → identifier | identifer, <ident_list>

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

Example of Context free grammar

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

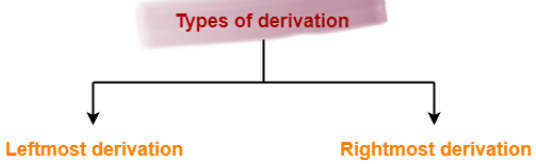
By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

DERIVATION

What is parse tree?

The process of deriving a string is called as derivation.

The geometrical representation of a derivation is called as a parse tree or derivation tree.



Leftmost Derivation-

•The process of deriving a string by expanding the leftmost non-terminal at each step is called as **leftmost derivation**.

Rightmost Derivation-

•The process of deriving a string by expanding the rightmost non-terminal at each step is called as **rightmost derivation**.

In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

Consider the grammar-

S → A1B
A → 0A / ∈
B → 0B / 1B / ∈

For the string w = 00101, find-
1.Leftmost derivation
2.Rightmost derivation
3.Parse Tree

Solution :

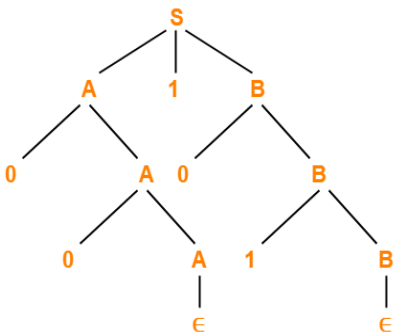
Leftmost Derivation-

S → **A**1B
→ 0**A**1B (Using A → 0A)
→ 00**A**1B (Using A → 0A)
→ 001**B** (Using A → ∈)
→ 0010**B** (Using B → 0B)
→ 00101**B** (Using B → 1B)
→ 00101 (Using B → ∈)

Rightmost Derivation-

S → A1B
→ A10B (Using B → 0B)
→ A101B (Using B → 1B)
→ A101 (Using B → ∈)
→ 0A101 (Using A → 0A)
→ 00A101 (Using A → 0A)
→ 00101 (Using A → ∈)

Parse Tree



Parse Tree

Ambiguity in Grammars

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

Semantics
Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.
CFG + semantic rules = Syntax Directed Definitions

For example:
int a = "value";
should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs.

The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Example:
$E \rightarrow E + T \{ E.value = E.value + T.value \}$

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

The attributes can be a number, type, memory location, return type etc….
Types of attributes are:
1. Synthesized attribute
2. Inherited attribute

Synthesized attributes

- Value of synthesized attribute at a node can be computed from the value of attributes at the children of that node in the parse tree.
- A syntax directed definition that uses synthesized attribute exclusively is said to be S-attribute definition.

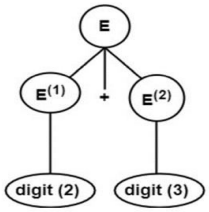
For Example, In Productions.

$E \rightarrow E^{(1)} + E^{(2)}$

$E \rightarrow$ digit

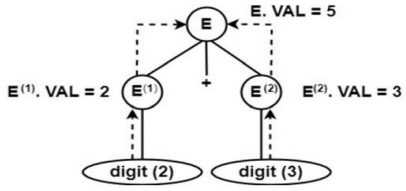
where digit = 0 | 1 | 2 |… . . | 9

The Parse Tree for string 2 + 3 will be



The complete parse tree, i.e., Parse Tree with translation will consist of a tree with each node labeled with attribute VAL.

Synthesized Translation



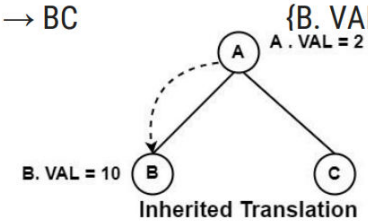
Here, E. VAL represents the value of non-terminal E, computed from the values at the children of that node in Parse Tree.

Inherited attribute

- An inherited value at a node in a parse tree is computed from the value of attributes at the parent and/or siblings of the node.

For example 1, a complete parse tree will be

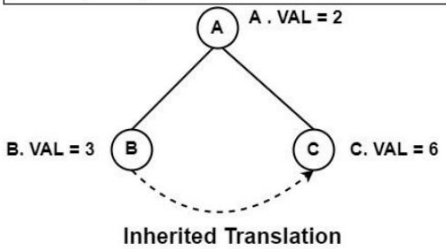
$A \rightarrow BC$ $\{B.VAL = 5 * A.VAL\}$



If A.VAL = 2, so, B.VAL = 5 * 2 = 10

So, the value of B is computed from the parent of that node.

Example 2 of Inherited Translation- Consider the production
$A \rightarrow BC$ {C. VAL = 2 * B. VAL}
Its complete parse tree will be



Here, value at node C on its sibling B. So, in inherited translation, the value of inherited attributes at a node depends upon its parent node or Sibling node.

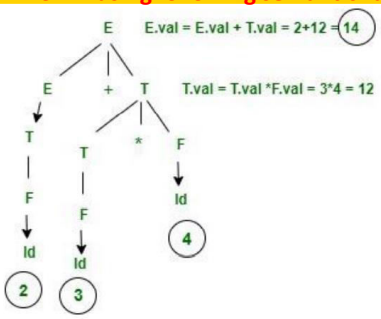
Example

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.
num.lexval is the attribute returned by the lexical analyzer.

Q: Draw the parse tree and translate the expression 2 + 3 * 2 using following semantic rule.

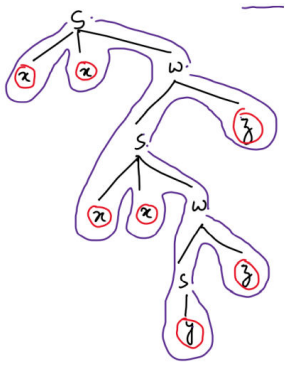
Production	Semantic Rules
E → E + T	{E.VAL := E.VAL + T.VAL }
E → T	{E.VAL := T.VAL }
T → T * F	{T.VAL := T.VAL * F. VAL }
T→ F	{T.VAL := F.VAL }
F → Id	{F.VAL := Id }



Write the value of xxxxyzz from the following translation scheme.

Production	Semantic Rules (Action)
S→ xxw	{print(1)}
S → y	{print(2)}
W→Sz	{print(3)}

Solution:
- Draw the Parse tree according to grammar then traverse through depth first traversal.
- Wherever you find the reduction, see the production and take the action.



Output is 23131

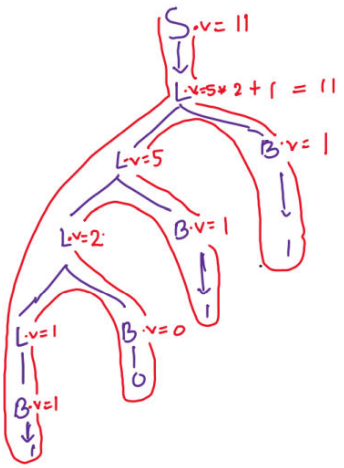
Convert 1011 into decimal number using following translation scheme.

Production	Semantic Rules
S → L	{S.VAL:= L.VAL }
L→ LB	{L.VAL := L.VAL * 2 + B.VAL }
L→ B	{L.VAL := B.VAL }
B→ 0	{B.VAL := 0 }
B→ 1	{B.VAL := 1 }

Solution:

- Draw the Parse tree according to grammar then traverse through depth first traversal.
- Wherever you find the reduction, see the production and take the action.

Output is 11



Any Questions?

Concepts

Introduction

Primitive Data Types

Array Types

CONCEPTS

- Introduction
- Names
- Variables
- The concept of binding
- Scope
- Scope and lifetime
- Referencing Environments
- Named constants

Introduction

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

Primitive Data Types

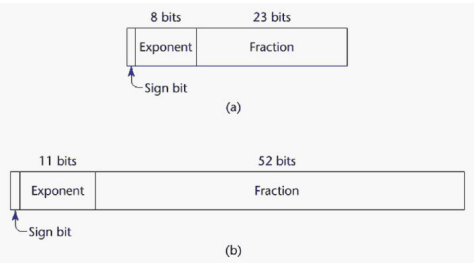
- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

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-Point Standard 754



Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, 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 (UCS-2)
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

Array Types

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

Array Design Issues

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

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements

  `array_name (index_value_list) → an element`

- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Most other languages use brackets

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- Index range checking

  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking
  - In Ada, the default is to require range checking, but it can be turned off

Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency

Subscript Binding and Array Categories (continued)

- *Stack-dynamic*: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)
- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

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 program execution)

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```


Heterogeneous Arrays

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

Array Initialization

- C-based languages
  - `int list [] = {1, 3, 5, 7}`
  - `char *names [] = {"Mike", "Fred","Mary Lou"};`
- Ada
  - `List : array (1..5) of Integer :=`
    `(1 => 17, 3 => 34, others => 0);`
- Python
  - `List comprehensions`

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

puts `[0, 9, 36, 81]` in `list`

Arrays Operations

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

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

- FORTRAN 95 is not: parameters, `EQUIVALENCE`
- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Ada is, almost (`UNCHECKED CONVERSION` is loophole)

  (Java and C# are similar to Ada)

Strong Typing (continued)

- 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

Names

- Design issues for names:
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names (continued)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN 95: maximum of 31
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C#, Ada, and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Names (continued)

- Special characters
  - PHP: all variable names must begin with dollar signs
  - Perl: all variable names begin with special characters, which specify the variable's type
  - Ruby: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Names (continued)

- Special words
  - An aid to readability; used to delimit or separate statement clauses
    - A *keyword* is a word that is special only in certain contexts, e.g., in Fortran
      - `Real VarName` (`Real` *is a data type followed with a name, therefore* `Real` *is a keyword)*
      - `Real = 3.4` (*`Real` is a variable)*
  - A *reserved word* is a special word that cannot be used as a user-defined name
  - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables

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

Variables Attributes

- Name - not all variables have them
- Address - the memory address with which it is associated
  - 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 are called aliases
  - Aliases are created via pointers, reference variables, C and C++ unions
  - Aliases are harmful to readability (program readers must remember all of them)

Variables Attributes (continued)

- *Type* - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
- *Value* - the contents of the location with which the variable is associated

  - The l-value of a variable is its address

  - The r-value of a variable is its value
- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

The Concept of Binding

A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol

- *Binding time* is the time at which a binding takes place.

Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a C or C++ `static` variable to a memory cell)
- Runtime -- bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

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

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)

Variable Attributes: 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 declared there
- The scope rules of a language determine how references to names are associated with 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*; the nearest static ancestor is called a *static parent*
- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Fortran 2003, and PHP)

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name
- Ada allows access to these "hidden" variables
  - E.g., **`unit.name`**

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Example in C:

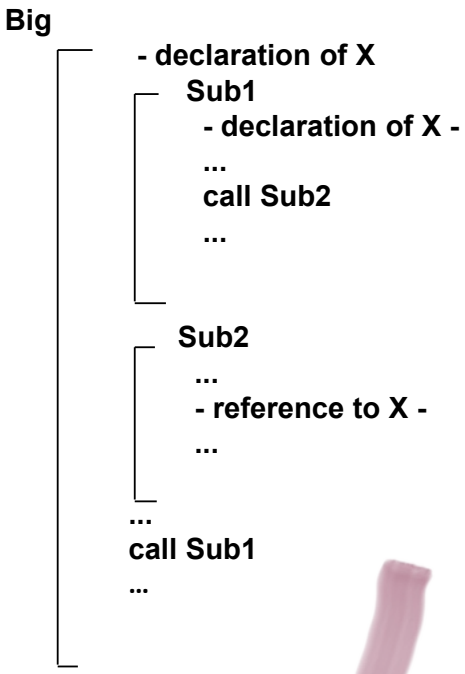
```
void sub() {
  int count;
  while (...) {
      int count;
    count++;
    ...
  }
  …
}
```

- Note: legal in C and C++, but not in Java and C# - too error-prone

Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions
- C and C++have both declarations (just attributes) and definitions (attributes and storage)
  - A declaration outside a function definition specifies that it is defined in another file

Scope Example



Big calls Sub1
Sub1 calls
Sub2
Sub2 uses X

Scope Example

- Static scoping
  - Reference to X is to Big's X
- Dynamic scoping
  - Reference to X is to Sub1's X
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically type check
    3. Poor readability- it is not possible to statically determine the type of a variable

Scope and Lifetime

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

Array Programs

An array is defined as the collection of similar type of data items stored at contiguous memory locations.

C program to sort the array of elements in ascending order using bubble sort method.

```
#include<stdio.h>
void main ()
{
   int i, j,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(i = 0; i<10; i++)
   {
      for(j = i+1; j<10; j++)
      {
         if(a[j] > a[i])
         {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         }
      }
   }
```


Array Programs

```
   printf("Printing Sorted Element List ...\n");
   for(i = 0; i<10; i++)
   {
      printf("%d\n",a[i]);
   }
}
```

Array Programs

Java Program to demonstrate the addition of two matrices in Java

```
class Testarray5
{
   public static void main(String args[])
   {
        //creating two matrices
        int a[][]={{1,3,4},{3,4,5}};
        int b[][]={{1,3,4},{3,4,5}};
        //creating another matrix to store the sum of two matrices
        int c[][]=new int[2][3];
        //adding and printing addition of 2 matrices
        for(int i=0;i<2;i++)
        {
             for(int j=0;j<3;j++)
             {
                  c[i][j]=a[i][j]+b[i][j];
                  System.out.print(c[i][j]+" ");
                             }
             System.out.println();//new line
        }
    }
}
```


Scope

Scope of Variable

Each variable is defined and can be used within its scope and determines that wherein the program this variable is available to use. The scope means the lifetime of that variable. It means the variable can only be accessed or visible within its scope.

Scope of goto

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.

// C program to check if a number is even or odd using goto statement

Scope

```
#include <stdio.h>
void checkEvenOrNot(int num) // function to check even or not
{
   if (num % 2 == 0)
      goto even;  // jump to even
  else
      goto odd;  // jump to odd
even:
   printf("%d is even", num);
   return; // return if even
odd:
   printf("%d is odd", num);
}
int main()
{
   int num = 26;
   checkEvenOrNot(num);
   return 0;
```


Scope

Local Vs. Global in Python

```
v1 = "Hey, I am Global Variable."   #globalvariable
def func1():
   v2="Hey, I am Local Variable." #localvariable
   print(v2)
func1()    #calling func1

def func2():
   print(v1)
func2()    #callin func2
```

Output:

Hey, I am a Local Variable
Hey, I am Global Variable

In the above program, we have taken one global variable v1 and one local variable v2. Since v1 is global, it can be easily accessed in any function, and v2 is local; it is used only within its declared function. But if we try to use v1 in func1, it will give an error.

Scope

Scope in Java

Local Vs. Global Variable in Java

In Java, there is no concept of global variables; since Java is an Object-oriented programming language, everything is a part of the Class. But if we want to make a variable globally accessible, we can make it static by using a **static** Keyword.

```
class Demo
{
        // static variable
        static int a = 10;
        // non-static or local variable
        int b = 20;
}
```


Scope

```
public class Main
 {
        public static void main(String[] args)
        {
                Demo obj = new Demo();
               // accessing the non-static variable
                System.out.println("Value of non-
static variable is: " + (obj.b));
                // accessing the static variable

        System.out.println("Value of static variable is:" + (Demo.a));
        }
}
```

Scope

Output:

Value of non-static variable is: 20
Value of static variable is:10

In the above program, we have used one local variable or non-static variable and one static variable. The local variable can be accessed by using the object of the Demo class, whereas the static variable can be accessed using the name of the class.

Expressions & Statements

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Introduction

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

Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls

Arithmetic Expressions: Design Issues

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

Arithmetic Expressions: Operators

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

Arithmetic Expressions: Operator Precedence Rules

- The *operator precedence rules* for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated
- Typical precedence levels
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -

Arithmetic Expressions: Operator Associativity Rule

- The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., in FORTRAN)
- APL is different; all operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overriden with parentheses

Ruby Expressions

- All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods
  - One result of this is that these operators can all be overriden by application programs

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - 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
```


Arithmetic Expressions: Operand Evaluation Order

- *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
  4. The most interesting case is when an operand is a function call

Arithmetic Expressions: 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 one-way parameters and lack of non-local references
  2. Write the language definition to demand that operand evaluation order be fixed
     - **Disadvantage**: limits some compiler optimizations
     - Java requires that operands appear to be evaluated in left-to-right order

Overloaded Operators

- Use of an operator for more than one purpose is called *operator overloading*
- Some are common (e.g., + for `int` and `float`)
- Some are potential trouble (e.g., `*` in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability

Overloaded Operators (continued)

- C++ and C# allow user-defined overloaded operators
- Potential problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense

Type Conversions

- A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type e.g., `float` to `int`
- A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., `int` to `float`

Type Conversions: Mixed Mode

- A *mixed-mode expression* is one that has operands of different types
- A *coercion* is an implicit type conversion
- Disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- In Ada, there are virtually no coercions in expressions

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - `C: (int)angle`
  - `Ada: Float (Sum)`

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

Type Conversions: Errors in Expressions

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

Relational and Boolean Expressions

- Relational Expressions
  - Use relational operators and operands of various types
  - Evaluate to some Boolean representation
  - Operator symbols used vary somewhat among languages (`!=`, `/=`, `~=`, `.NE.`, `<>`, `#`)
- JavaScript and PHP have two additional relational operator, `===` and `!==`

  - Similar to their cousins, == and `!=`, except that they do not coerce their operands

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators

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

Relational and Boolean Expressions: No Boolean Type in C

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

Short Circuit Evaluation

- An expression in which the result is determined without evaluating all of the operands and/or operators
- Example: **`(13*a) * (b/13-1)`**
  If `a` is zero, there is no need to evaluate **`(b/13-1)`**
- Problem with non-short-circuit evaluation

```
index = 1;
while (index <= length) && (LIST[index] != value)
        index++;
```

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

Short Circuit Evaluation (continued)

- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (**`&&`** and **`||`**), but also provide bitwise Boolean operators that are not short circuit (**`&`** and **`|`**)
- Ada: programmer can specify either (short-circuit is specified with **`and then`** and **`or else`**)
- Short-circuit evaluation exposes the potential problem of side effects in expressions
  e.g. **`(a > b) || (b++ / 3)`**

Assignment Statements

- The general syntax
  `<target_var> <assign_operator> <expression>`
- The assignment operator
  = FORTRAN, BASIC, the C-based languages
  := ALGOLs, Pascal, Ada
- = can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

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

Which is equivalent to

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


Assignment Statements: Compound Operators

- 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
```

Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
- Examples

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

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

  `count++` (`count` incremented)

  `-count++` (`count` incremented then negated)

Assignment as an Expression

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

```
while ((ch = getchar())!=
EOF){…}
```

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

List Assignments

- Perl and Ruby support list assignments

  e.g.,

```
($first, $second, $third) = (20, 30, 40);
```


Mixed-Mode Assignment

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

Summary

- Expressions
- Operator precedence and associativity
- Operator overloading
- Mixed-type expressions
- Various forms of assignment

Example Programs related to expressions in various programming languages:

C program to check the entered age is suitable to married or not using conditional operator.

```
#include<stdio.h>
#include<string.h>
int main()
{
   int age = 25;
   char status;
   status = (age>22) ? 'M': 'U';
   if(status == 'M')
   printf("Married");
   else
   printf("Unmarried");
   return 0;
}
```

Output

Married

Java program to perform Arithmetic Expression

```
public class OperatorExample
{
    public static void main(String args[])
    {
            System.out.println(10*10/5+3-1*4/2);
    }
}
```

Output:

Java program to show OR Operator example: Logical || and Bitwise |

```
public class OperatorExample
{
    public static void main(String args[])
    {
            int a=10;
            int b=5;
            int c=20;
            System.out.println(a>b||a<c); //true || true = true
            System.out.println(a>b|a<c);  //true | true = true
            System.out.println(a>b||a++<c);       //true || true = true
            System.out.println(a);            //10 because second condition is not checked
            System.out.println(a>b|a++<c);        //true | true = true
            System.out.println(a);            //11 because second condition is checked
    }
}
```

Python program to perform logical expressions

```
P = (10 == 9)
Q = (7 > 5)

# Logical Expressions
R = P and Q
S = P or Q
T = not P

print(R)
print(S)
print(T)
```

output:

```
False
True
True
```

CONCEPTS

- ***Introduction***
- ***Fundamentals of Subprograms***
- ***Design Issues for Subprograms***
- ***Local Referencing Environments***
- ***Parameter-Passing Methods***
- ***Parameters That Are Subprograms***
- ***Overloaded Subprograms***
- ***Generic Subprograms***
- ***Design Issues for Functions***
- ***User-Defined Overloaded Operators***
- ***Coroutines***

CONCEPTS

- ***The General Semantics of Calls and Returns***
- ***Implementing "Simple" Subprograms***
- ***Implementing Subprograms with Stack-Dynamic Local Variables***
- ***Nested Subprograms***
- ***Blocks***
- ***Implementing Dynamic Scoping***

Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
  - Data abstraction
    - Emphasized in the1980s

Fundamentals of Subprograms

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

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
  - In Python, function definitions are executable; in all other languages, they are non-executable
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type

Basic Definitions (continued)

- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

- Positional
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter's names

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated
- Variable numbers of parameters
  - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**
  - In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
  - In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
  - In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods

Ruby Blocks

- Ruby includes a number of iterator functions, which are often used to process the elements of arrays
- Iterators are implemented with blocks, which can also be defined by applications
- Blocks are attached methods calls; they can have parameters (in vertical bars); they are executed when the method executes a **`yield`** statement

```
def fibonacci(last)
   first, second = 1, 1
   while first <= last
     yield first
     first, second = second, first + second
   end
 end

 puts "Fibonacci numbers less than 100 are:"
 fibonacci(100) {|num| print num, " "}
 puts
```


Procedures and Functions

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

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprogram be generic?

Local Referencing Environments

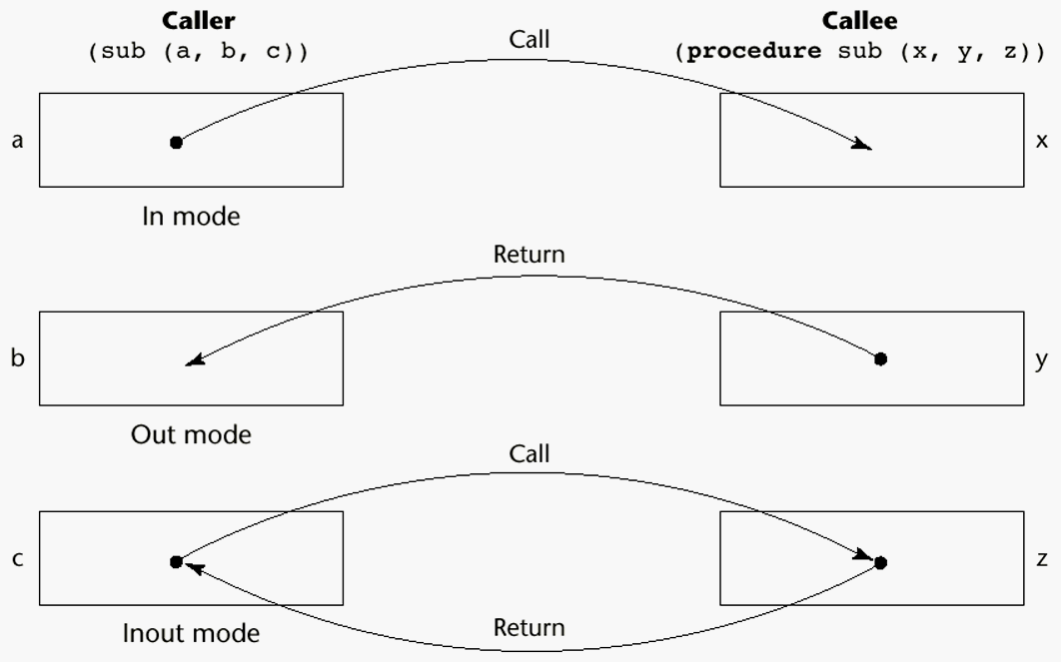
- Local variables can be stack-dynamic

  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages
    - Allocation/de-allocation, initialization time
    - Indirect addressing
    - Subprograms cannot be history sensitive
- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Conceptual Models of Transfer

- Physically move a path
- Move an access path

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Result (Out Mode)

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

Pass-by-Value-Result (inout Mode)

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

Pass-by-Reference (Inout Mode)

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

Pass-by-Name (Inout Mode)

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

Implementing Parameter-Passing Methods

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

Parameter Passing Methods of Major Languages

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All parameters are passed are passed by value
  - Object parameters are passed by reference
- Ada
  - Three semantics modes of parameter transmission: `in, out, in out; in` is the default mode
  - Formal parameters declared `out` can be assigned but not referenced; those declared `in` can be

Parameter Passing Methods of Major Languages (continued)

- Fortran 95
  - Parameters can be declared to be in, out, or inout mode
- C#
  - Default method: pass-by-value
    - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with `ref`
- PHP: very similar to C#
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects)

Multidimensional Arrays as Parameters

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

Multidimensional Arrays as Parameters: C and C++

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

Multidimensional Arrays as Parameters: Ada

- Ada – not a problem
  - Constrained arrays – size is part of the array's type
  - Unconstrained arrays - declared size is part of the object declaration

Multidimensional Arrays as Parameters: Fortran

- Formal parameter that are arrays have a declaration after the header
  - For single-dimension arrays, the subscript is irrelevant
  - For multidimensional arrays, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function

Multidimensional Arrays as Parameters: Java and C#

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

Overloaded Subprograms

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

Examples of parametric polymorphism: C++

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

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

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

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
  - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
  - Lua allows functions to return multiple values

User-Defined Overloaded Operators

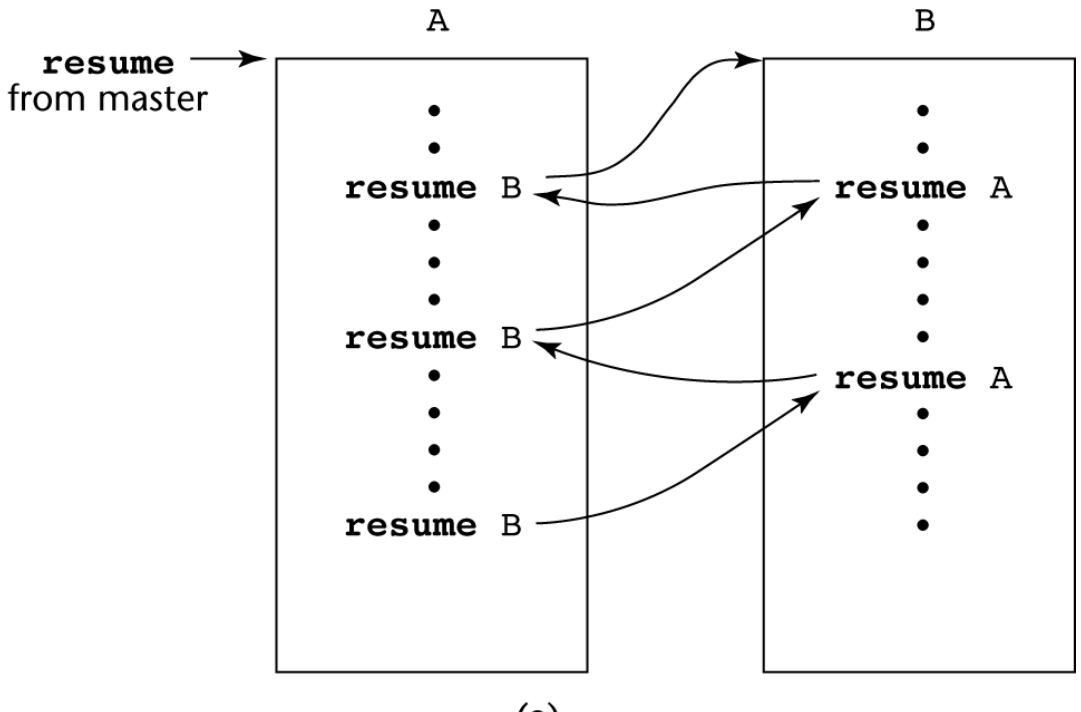
- Operators can be overloaded in Ada, C++, Python, and Ruby
- An Ada example

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

Coroutines

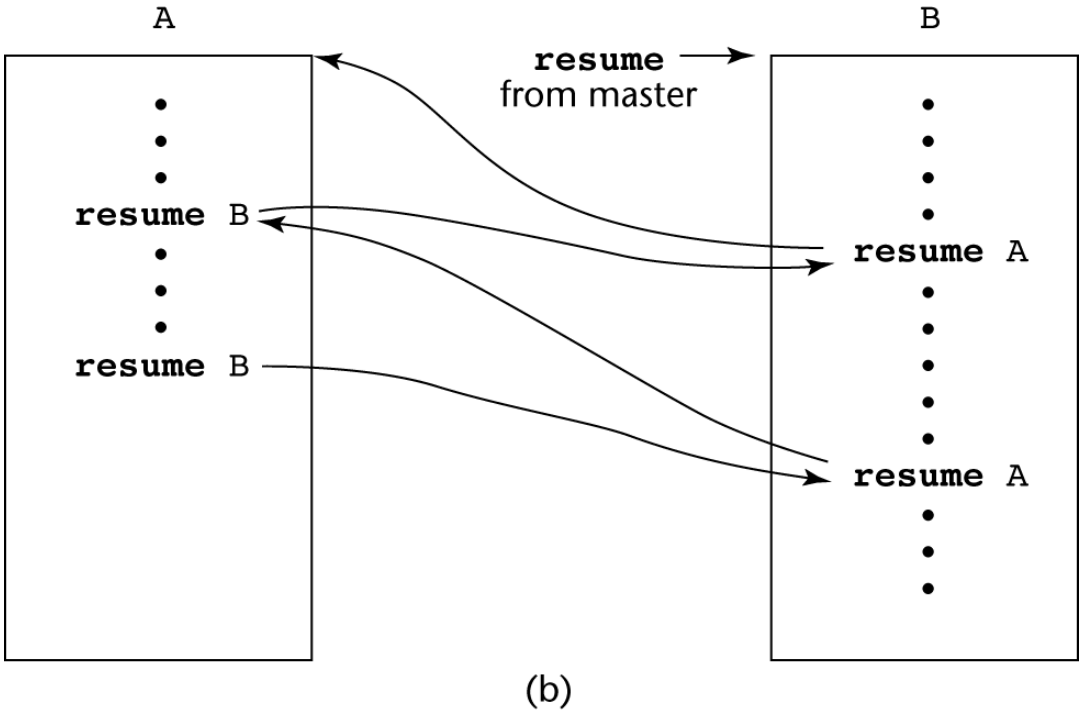
- A *coroutine* is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent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

Coroutines Illustrated: Possible Execution Controls



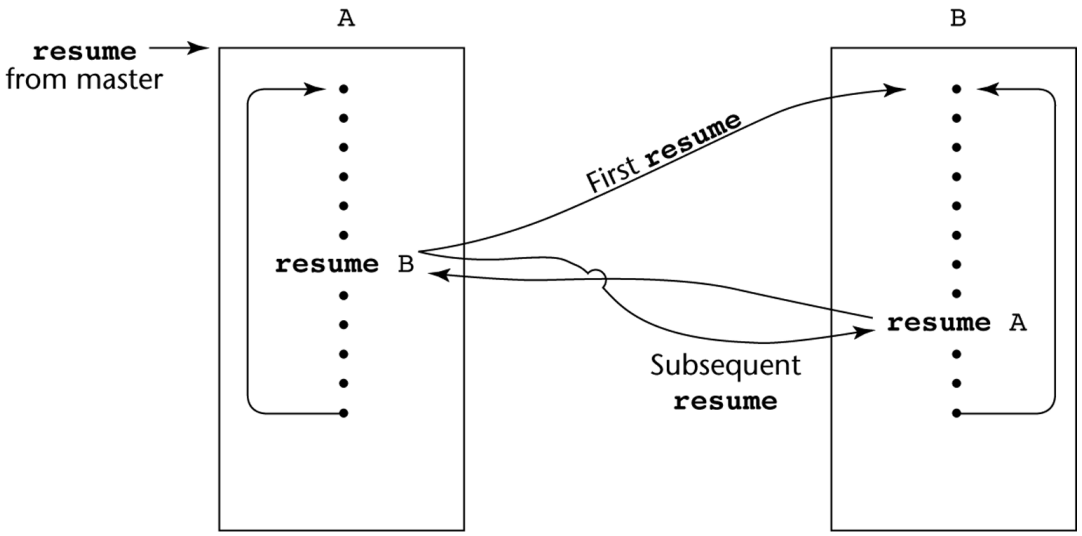
(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops



C program for function with argument and with return value

```
#include<stdio.h>
int sum(int, int);
void main()
{
   int a,b,result;
   printf("\nGoing to calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   result = sum(a,b);
   printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
   return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:
Enter two numbers:10
20
The sum is : 30
```

Java program function to check the number is odd or even using runtime input

```
import java.util.Scanner;
public class EvenOdd
{
    public static void main (String args[])
    {
            //creating Scanner class object
            Scanner scan=new Scanner(System.in);
            System.out.print("Enter the number: ");
            //reading value from user
            int num=scan.nextInt();
            //method calling
            findEvenOdd(num);
    }
    //user defined method
    public static void findEvenOdd(int num)
    {
            //method body
            if(num%2==0)
                    System.out.println(num+" is even");
            else
                    System.out.println(num+" is odd");
        }
}
```

Output 1:

```
Enter the number: 12
12 is even
```

Output 2:

```
Enter the number: 99
99 is odd
```

Python code to demonstrate the use of return statements

```
# Defining a function with return statement
def square( num ):
   return num**2

# Calling function and passing arguments.
print( "With return statement" )
print( square( 39 ) )

# Defining a function without return statement
```

```
def square( num ):
    num**2

# Calling function and passing arguments.
print( "Without return statement" )
print( square( 39 ) )
```

Output:

```
With return statement
1521
Without return statement
None
```

Chapter 6

Functional and Logical 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) ≡ 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

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

- Inefficient execution
- Programs can automatically be made concurrent

Logic Programming Languages

Topics

- Introduction
- A Brief Introduction to Predicate Calculus
- Predicate Calculus and Proving Theorems
- An Overview of Logic Programming
- The Origins of Prolog
- The Basic Elements of Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

Introduction

- *Logic* programming languages, sometimes called *declarative* programming languages
- 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 producing them)

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 *predicate calculus*

Object Representation

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

• *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	¬	¬ a	not a
conjunction	∩	a∩ b	a and b
disjunction	∪	a∪ b	a or b
equivalence	≡	a≡ b	a is equivalent to b
implication	⊃ ⊂	a⊃ b a⊂ 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 from known 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 matching process to succeed

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

• 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 the form 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 an uppercase letter
- *Instantiation*: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal
- *Structure*: represents atomic proposition
  functor(*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 system to prove or disprove – *goal statement*
- Same format as headless Horn

```
man(fred)
```

- Conjunctive propositions and propositions with variables also legal goals

```
father(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 sequence that 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 facts in 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 before working 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 every subgoal

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)

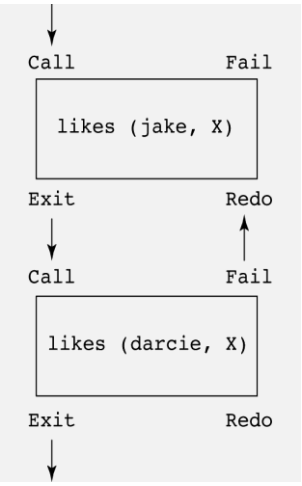
—*Redo* (when backtrack occurs)

—*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 already seen): 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