**UNIT - I**

1. **What are the reasons for studying concepts of Programming languages?**

**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?

**2.Briefly discuss a few of the areas of computer applications and their associated languages.**

**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

**3. Write an evaluation of some programming language you know using the criteria for a good programming language?**

**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 fromreadability. 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 leastpartially 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 orduring 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 thencontinue is a great aid to reliability.
* **Aliasing**: it is having two or more distinct referencing methods, or names, for the same memorycell.

– 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)

**4. Explain about the von Newmann computer architecture.**

**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

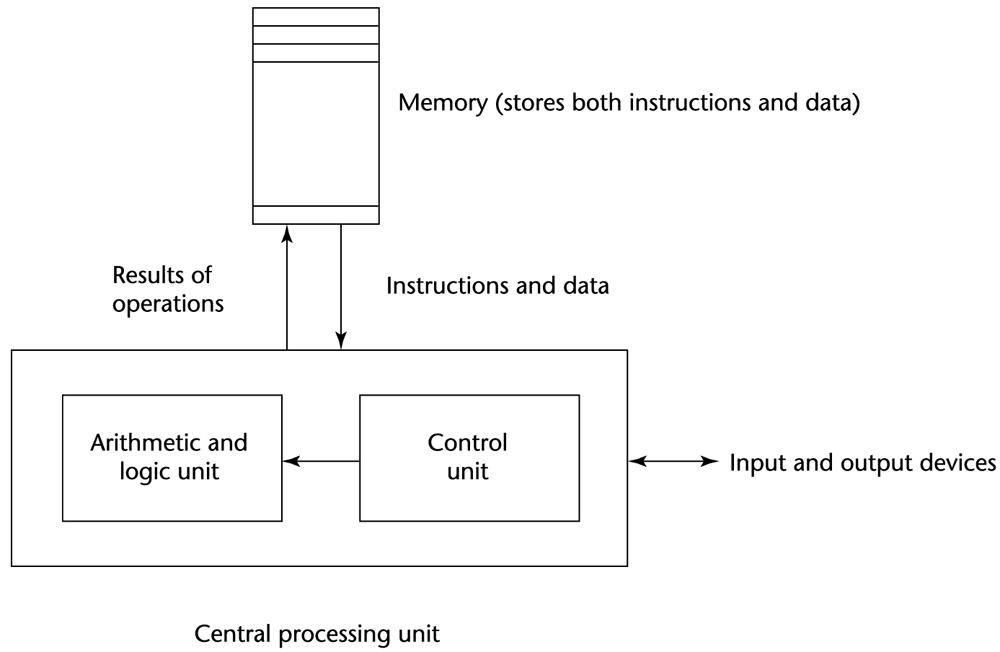
– Memory is separate from CPU

– 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



**4.Write about 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

**5. Explain the language categories with example?**

**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

**6. Explain about Programming Environments ?**

* The collection of tools used in software development
* **UNIX**

– An older operating system and tool collection

* **Borland JBuilder**

– An integrated development environment for Java

* **Microsoft Visual Studio.NET**

– A large, complex visual environment

– Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

7.7.

**7.Define syntax and semantics?**

**Syntax and Semantics**

**Syntax** - theform or structure of the expressions, statements, and program units **Semantics -** the meaning of the expressions, statements, and program units

**8.Write about Formal approaches to describing syntax?**

1**. Recognizers** - used in compilers

2**. 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 all practical 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.

**9. Define CFG? write about BNF and EBNF method?**

**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 form is 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>

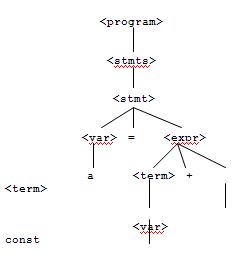
=> a = b + const

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



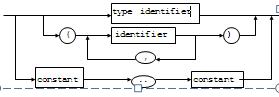
**EBNF**:

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

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

**Syntax Graphs** - put the terminals in circles or ellipses and put the nonterminals in rectangles; connectwith 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
   1. 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 X0 -> X1 ... Xn be a rule Functions of the form S(X0) = f(A(X1), ... A(Xn)) define *synthesized attributes* Functions of the form I(Xj) = f(A(X0), ... , A(Xn)), for i <= j <= n, define *inherited attributes* Initially, there are *intrinsic attributes* on the leaves

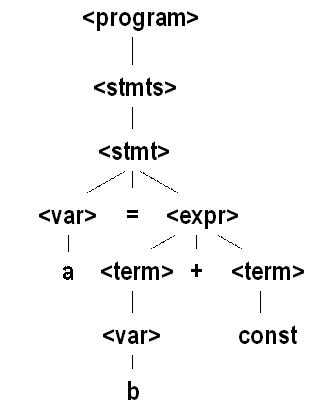
**Example***:*expressions of the form id + id - id's can be either int\_type or real\_type

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

**10.Define Parse Tree? Explain with an example?**

**PARSE TREE:**

1. 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



**11. What is an attribute grammar? Explain with example?**

**Ambiguous grammar**

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

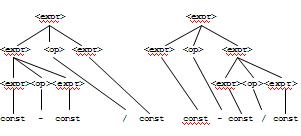
<expr> -> <expr> <op> <expr> | const

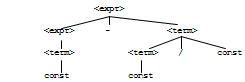
<op> -> / | -

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

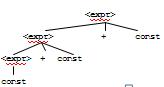
=> const - const / const

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>

**Attribute Grammar**

**Attributes:**

actual\_type - synthesized for <var> and <expr>

expected\_type - inherited for <expr>

**Attribute Grammar:**

1. Syntax rule: <expr> -> <var>[1] + <var>[2] Semantic rules:

<var>[1].env  <expr>.env

<var>[2].env  <expr>.env

<expr>.actual\_type  <var>[1].actual\_type

Predicate:

<var>[1].actual\_type = <var>[2].actual\_type

<expr>.expected\_type = <expr>.actual\_type

1. Syntax rule: <var> -> id Semantic rule:

<var>.actual\_type <- lookup (id, <var>.env)

**12.How are attribute values computed?**

1. If all attributes were inherited, the tree could be decorated in top-down order.

1. If all attributes were synthesized, the tree could be decorated in bottom-up order.
2. In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used

. <expr>.env  inherited from parent

<expr>.expected\_type  inherited from parent

<var>[1].env  <expr>.env

<var>[2].env  <expr>.env

<var>[1].actual\_type  lookup (A, <var>[1].env)

<var>[2].actual\_type  lookup (B, <var>[2].env)

<var>[1].actual\_type =? <var>[2].actual\_type

1. <expr>.actual\_type  <var>[1].actual\_type <expr>.actual\_type =? <expr>.expected\_type

**13. Explain about Dynamic Semantics?**

* No single widely acceptable notation or formalism for describing semantics

I. **Operational Semantics**

- Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement

* + - * To use operational semantics for a high-level language, a virtual machine in needed
      * A *hardware* pure interpreter would be too expensive
      * A *software* pure interpreter also has problems:
    1. The detailed characteristics of the particular computer would make actions difficult to understand
  1. Such a semantic definition would be machine dependent
* *A better alternative*: A complete computer simulation
* ***The process****:*
  + 1. Build a translator (translates source code to the machine code of an idealized computer)
    2. Build a simulator for the idealized computer
* ***Evaluation of operational semantics****:*
* Good if used informally
* Extremely complex if used formally (e.g., VDL

**Axiomatic Semantics**

* + Based on formal logic (first order predicate calculus)
* ***Original purpose****:*formal program verification
* ***Approach:*** Define axioms or inference rules for each statement type in the language(to allow transformations of expressions to other expressions)
* The expressions are called ***assertions***

**14.Explain the Denotational Semantics? And how they can evaluate?**

* Based on recursive function theory
* The most abstract semantics description method
* Originally developed by Scott and Strachey
* The process of building a denotational spec for a language:
  1. Define a mathematical object for each language entity
  2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
* The meaning of language constructs are defined by only the values of the program's variables

- The difference between denotational and operational semantics: In operational semantics, the state

changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous

mathematical functions

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

s = {<i1, v1>, <i2, v2>,…, <in, vn>}

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

1. Decimal Numbers

<dec\_num>  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <dec\_num> (0 | 1 | 2 | 3 | 4 |

5 | 6 | 7 | 8 | 9)

Mdec ('0') = 0, Mdec ('1') = 1, …, Mdec ('9') = 9

Mdec (<dec\_num> '0') = 10 \* Mdec (<dec\_num>)

Mdec (<dec\_num> '1’) = 10 \* Mdec (<dec\_num>) + 1

…

Mdec (<dec\_num> '9') = 10 \* Mdec (<dec\_num>) + 9

1. Expressions Me(<expr>, s) =

case <expr> of

<dec\_num> => Mdec(<dec\_num>, s) <var> =>

if VARMAP(<var>, s) = undef then error

else VARMAP(<var>, s)

<binary\_expr> =>

if (Me(<binary\_expr>.<left\_expr>, s) = undef

OR Me(<binary\_expr>.<right\_expr>, s) =

undef)

then error

else

if (<binary\_expr>.<operator> = ë+í then

Me(<binary\_expr>.<left\_expr>, s) +

Me(<binary\_expr>.<right\_expr>, s)

else Me(<binary\_expr>.<left\_expr>, s) \*

Me(<binary\_expr>.<right\_expr>, s)

3 Assignment Statements

Ma(x := E, s) =

if Me(E, s) = error

then error

else s’ = {<i1’,v1’>,<i2’,v2’>,...,<in’,vn’>},

where for j = 1, 2, ..., n,

vj’ = VARMAP(ij, s) if ij <> x

* Me (E, s) if ij = x

4 Logical Pretest Loops Ml(while B do L, s) =

if Mb(B, s) = undef then error

else if Mb(B, s) = false then s

else if Msl (L, s) = error then error

else Ml(while B do L, Msl(L, s))

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

 In essence, the loop has been converted from iteration to recursion, where the recursive control

is mathematically defined by other recursive state mapping functions

 Recursion, when compared to iteration, is easier to describe with mathematical rigor

**Evaluation of denotational semantics:**

* Can be used to prove the correctness of programs
* Provides a rigorous way to think about programs
* Can be an aid to language design
* Has been used in compiler generation systems

**UNIT-II**

**DATA TYPES**

**1)Define Data type?and types of datatypes?**

**Data type Definition:**

– collection of data objects

– a set of predefined operations

* **descriptor** :collection of attributes for a variable
* **object** :instance of a user-defined (abstract data) type

**Data Types:**

* **Primitive**

– not defined in terms of other data types

– defined in the language

– often reflect the hardware

* **Structured**

– built out of other types

**Integer Types:**

* + Usually based on hardware
    - May have several ranges

– **Java’s signed integer sizes**: byte, short, int, long

– **C/C++** have unsigned versions of the same types

– **Scripting languages** often just have one integer type

– **Python** has an integer type and a long integer which can get as big as it needs to.

1. **Representing Integers:**

– Can convert positive integers to base

– How do you handle negative numbers with only 0s and 1s?

– Sign bit

– Ones complement

– Twos complement - this is the one that is used

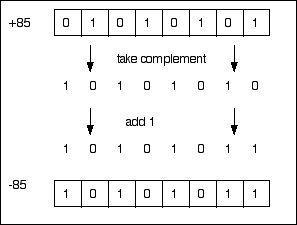
– Representing negative integers.

**Representing negative integers:**

* Sign bit
* Ones complement

**Twos Complement:**

* To get the binary representation, take the complement and add 1



**Floating Point Types:**

* Model real numbers

only an approximation due to round-off error

* For scientific use support at least two floating-point types (e.g., float and double; sometimes more)
* The float type is the standard size, usually being stored in four bytes of memory.
* The double type is provided for situations where larger fractional parts and/or a larger range of exponents is needed
* Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation
* The collection of values that can be represented by a floating-point type is defined in terms of precision and range
* **Precision** is the accuracy of the fractional part of a value, measured as the number of bits
* **Range** is a combination of the range of fractions and, more important, the range of exponents.
* Usually based on hardware
* IEEE Floating-Point Standard 754

–32 and 64 bit standards

**Representing Real Numbers:**

* We can convert the decimal number to base 2 just as we did for integers
* How do we represent the decimal point?

– fixed number of bits for the whole and fractional parts severely limits the range of values we can represent

* Use a representation similar to scientific notation

**IEEE Floating Point Representation:**

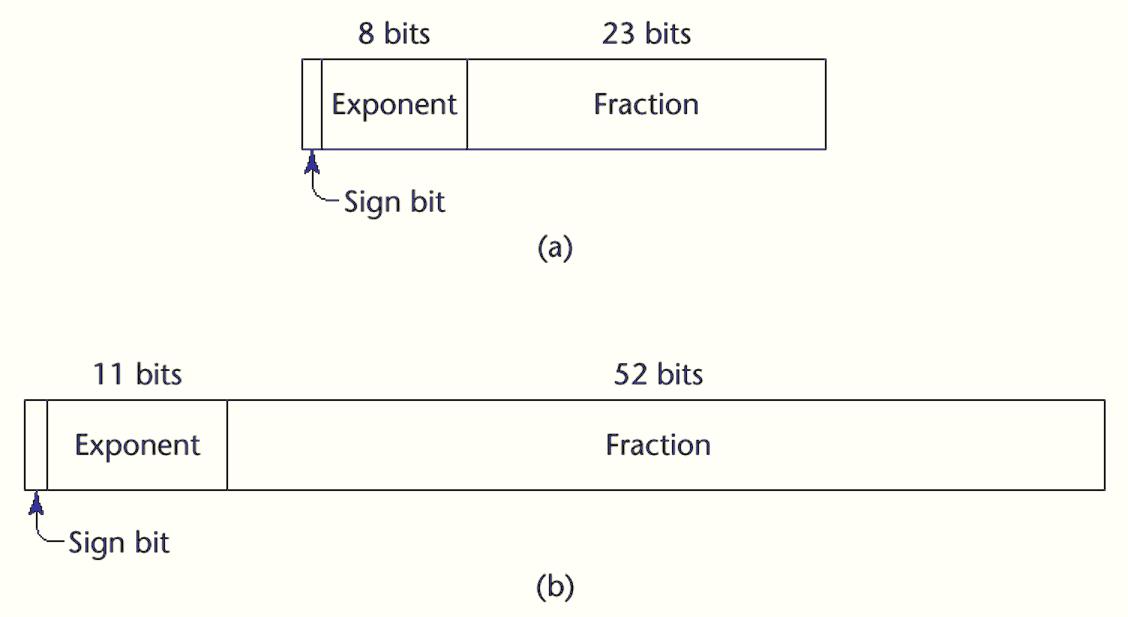
* Normalize the number

– one bit before decimal point

* Use one bit to represent the sign (1 for negative)
* Use a fixed number of bits for the exponent which is offset to allow for negative exponents

– Exponent = exponent + offset

* (-1)sign 1.Fraction x 2Exponent



**Floating Point Types:**

* C, C++ and Java have two floating point types

– float

– double

* Most scripting languages have one floating point type

– Python's floating point type is equivalent to a C double

* Some scripting languages only have one kind of number which is a floating point type

**Fixed Point Types (Decimal) :**

* For business applications (money) round-off errors are not acceptable

– Essential to COBOL

– .NET languages have a decimal data type

* Store a fixed number of decimal digits
* Operations generally have to be defined in software
* ***Advantage***: accuracy
* ***Disadvantages***: limited range, wastes memory

**C# decimal Type:**

* 128-bit representation
* **Range:** 1.0x10-28to 7.9x1028
* **Precision**: representation is exact to 28 or 29 decimal places (depending on size of number)

– no roundoff error

**Other Primitive Data Types:**

* **Boolean**

– Range of values: two elements, one for “true” and one for “false”

– Could be implemented as bits, but often as bytes

– Boolean types are often used to represent switches or flags in programs.

– A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.

* **Character**

– Stored as numeric codings

– Most commonly used coding: ASCII

– An alternative, 16-bit coding: Unicode

* Complex (Fortran, Scheme, Python)
* Rational (Scheme)

**Character Strings :**

* Values are sequences of characters
* Character string constants are used to label output, and the input and output of all kinds of data are often done in terms of strings.
* **Operations:**

– Assignment and copying

– Comparison (=, >, etc.)

– Catenation

– Substring reference

– Pattern matching

– A **substring reference** is a reference to a substring of a given string. Substring references are discussed in the more general context of arrays, where the substring references are called **slices**.

– In general, both assignment and comparison operations on character strings are complicated by the possibility of string operands of different lengths.

2)Write about character data type?

* **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 Implementations:**

* **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

* **Java**

– String class

**String Length Options**

* **Static**: COBOL, Java’s String class
* **Limited Dynamic Length**: C and C++

– a special character is used to indicate the end of a string’s characters

* **Dynamic (**no maximum): SNOBOL4, Perl, JavaScript
* Ada supports all three **string length options**

**String Implementation**

* **Static length:** compile-time descriptor
* **Limited dynamic length**:may need run-time descriptor

• not in C and C++

* **Dynamic length:** needs run-time descriptor;

– allocation/deal location is main implementation issue

**3)define user defined datatypes and explain with an example?**

**User-Defined Ordinal Types:**

* **ordinal type** :range of possible values corresponds to set of positive integers
* **Primitive ordinal types**

– integer

– char

– boolean

* **User-defined ordinal types**

– enumeration types

– subrange types

**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

– duplication of names

– coercion rules

**Enums in C (and C++):**

* To define an enumerated type in **C**

**Ex:**

**- Enum weekday {Sunday, Monday, Tuesday, Wednesday, Thursday,** **Friday, Saturday};**

**Enum weekday today = Tuesday;**

* Use **typedef** to give the type a name

**typedef enum weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} weekday;**

**Weekday today = Tuesday;**

* By default, values are consecutive starting from 0.

– You can explicitly assign values

**Enum months {January=1, February,};**

Enumerations in Java 1.5:

* An enum is a new class which extends **java.lang.Enum** and implements Comparable

– Get type safety and compile-time checking

– Implicitly public, static and final

– Can use either == or equals to compare

– toString and valueOf are overridden to make input and output easier

**Java enum Example:**

* **Defining an enum type**

Enum Season {WINTER, SPRING, SUMMER, FALL};

* **Declaring an enum variable**

Season season = Season. WINTER;

* to String gives you the **string representation of the name** System.out.println (season);
  + prints WINTER
* **valueOf l**ets you convert a String to an enum

Season = valueOf (“SPRING”);

**Sub range Types:**

* A contiguous subsequence of an ordinal type

**Example:**

* **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;

**Evaluation**

Subrange types enhance readability by making it clear to readers that variables of subtypes can store only certain ranges of values. Reliability is increased with subrange types, because assigning a value to a subrange variable that is outside the specified range is detected as an error, either by the compiler (in the case of the assigned value being a literal value) or by the run-time system (in the case of a variable or expression). It is odd that no contemporary language except Ada has subrange types.

**Implementation of User-Defined Ordinal Types**

* Enumeration types are implemented as integers
* Subrange types are implemented like the parent types

– code inserted (by the compiler) to restrict assignments to subrange variables

**4)Define an array and explain it’s design issues?**

**Arrays and Indices**

* Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the aggregate name, and the second part is a possibly dynamic selector consisting of one or more items known as **subscripts** or **indices**.
* If all of the subscripts in a reference are constants, the selector is static; otherwise, it is dynamic. The selection operation can be thought of as a mapping from the array name and the set of

subscript values to an element in the aggregate. Indeed, arrays are sometimes called **finite** **mappings**. Symbolically, this mapping can be shown as

* + array\_name(subscript\_value\_list) → element

**Subscript Bindings and Array Categories**

* The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound.
* There are five categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated.
* A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time).
  + - The **advantage** of static arrays is efficiency: No dynamic allocation or deallocation is required.
    - The **disadvantage i**s that the storage for the array is fixed for the entire execution time of the program.
* A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution.
  + - * + The **advantage** of fixed stack-dynamic arrays over static arrays is space efficiency
        + The **disadvantage** is the required allocation and deallocation time
* A **stack-dynamic array** is one in which both the subscript ranges and the storage allocation are dynamically bound at elaboration time. Once the subscript ranges are bound and the storage is allocated, however, they remain fixed during the lifetime of the variable.
  + - * + The **advantage** of stack-dynamic arrays over static and fixed stack-dynamic arrays is flexibility
* A **fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated
  + - * The **advantage** of fixed heap-dynamic arrays is flexibility—the array’s size always fits the problem.
      * The **disadvantage** is allocation time from the heap, which is longer than allocation time from the stack.
* A **heap-dynamic array** is one in which the binding of subscript ranges and storage allocation is

dynamic and can change any number of times during the array’s lifetime.

* The **advantage** of heap-dynamic arrays over the others is flexibility:
* The **disadvantage** is that allocation and deallocation take longer and may happen many times during execution of the program.

**Array Initialization**

Some languages provide the means to initialize arrays at the time their storage is allocated.

An array aggregate for a single-dimensioned array is a list of literals delimited by parentheses and slashes. For example, we could have

Integer, Dimension (3) :: List = (/0, 5, 5/)

In the C declaration

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

These arrays can be initialized to string constants, as in **char** name [] = "freddie";

Arrays of strings in C and C++ can also be initialized with string literals. In this case, the array is one of

pointers to characters.

For example,

**char** \*names [] = {"Bob", "Jake" ,"Darcie"};

In Java, similar syntax is used to define and initialize an array of references to String objects. For example,

String[] names = ["Bob", "Jake", "Darcie"];

Ada provides two mechanisms for initializing arrays in the declaration statement: by listing them in the

order in which they are to be stored, or by directly assigning them to an index position using the =>

operator, which in Ada is called an **arrow**.

For example, consider the following:

List : **array** (1.5) **of** Integer := (1, 3, 5, 7, 9);

Bunch : **array** (1.5) **of** Integer := (1 => 17, 3 => 34,

**others** => 0);

**Rectangular and Jagged Arrays**

A **rectangular array** is a multi dimensioned array in which all of the rows have the same number of elements and all of the columns have the same number of elements. Rectangular arrays model rectangular tables exactly.

A **jagged array** is one in which the lengths of the rows need not be the same.

For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements. This also applies to the columns and higher dimensions. So, if there is a third dimension (layers), each layer can have a different number of elements. Jagged arrays are made possible

when multi dimensioned arrays are actually arrays of arrays. For example, a matrix would appear as an

array of single-dimensioned arrays.

For example,

myArray[3][7]

**Slices**

A **slice** of an array is some substructure of that array.

For example, if A is a matrix, then the first row of A is one possible slice, as are the last row and the first column. It is important to realize that a slice is not a new data type. Rather ,it is a mechanism for referencing part of an array as a unit.

**Evaluation**

Arrays have been included in virtually all programming languages

**Implementation of Array Types**

Implementing arrays requires considerably more compile-time effort than does implementing primitive types. The code to allow accessing of array elements must be generated at compile time. At run time, this code must be executed to produce element addresses. There is no way to pre compute the address to be accessed by a reference such as

list [k]

A single-dimensioned array is implemented as a list of adjacent memory cells. Suppose the array list is defined to have a subscript range lower bound of 0. The access function for list is often of the form address

( list [k] ) = address (list [0] ) + k \* element\_size where the first operand of the addition is the constant part of the access function, and the second is the variable part .

If the element type is statically bound and the array is statically bound to storage, then the value of the constant part can be computed before run time. However, the addition and multiplication operations must be done at run time.

The generalization of this access function for an arbitrary lower bound is address (list[k]) = address ( list [lower\_bound]) + ( (k - lower\_bound) \* element\_size)

**5)what is an Associative Arrays and explain it’s operations?**

An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**. In the case of non-associative arrays, the indices never need to be stored (because of their regularity). In an associative array, however, the user-defined keys must be stored in the structure. So each element of an associative array is in fact a pair of entities, a key and a value. We use Perl’s design of associative arrays to illustrate this data structure. Associative arrays are also supported directly by

Python, Ruby, and Lua and by the standard class libraries of Java, C++, C#, and F#. The only design issue that is specific for associative arrays is the form of references to their elements.

**Structure and Operations**

In Perl, associative arrays are called **hashes**, because in the implementation their elements are stored and retrieved with hash functions. The namespace for Perl hashes is distinct: Every hash variable name must begin with a percent sign (%). Each hash element consists of two parts: a key, which is a string, and a value, which is a scalar (number, string, or reference). Hashes can be set to literal values with the assignment statement, as in

%salaries = ("Gary" => 75000, "Perry" => 57000, "Mary" => 55750, "Cedric" => 47850);

Recall that scalar variable names begin with dollar signs ($).

For example,

$salaries {"Perry"} = 58850;

A new element is added using the same assignment statement form. An element can be removed from the hash with the **delete** operator, as in

**Delete** $salaries{"Gary"};

**6)Define record and explain with an its implementation?**

**Record Types**

A **record** is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure. There is frequently a need in programs to model a collection of data in which the individual elements are not of the same type or size. For example, information about a college student might include name, student number, grade point average, and so forth. A data type for such a collection might use a character string for the name, an integer for the student number, a floating point for the grade point average, and so forth. Records are designed for this kind of need.

The following **design issues** are specific to records:

* What is the syntactic form of references to fields?
* Are elliptical references allowed?

**Definitions of Records**

The fundamental difference between a record and an array is that record elements, or **fields**, are not referenced by indices. Instead, the fields are named with identifiers, and references to the fields are made using these identifiers The COBOL form of a record declaration, which is part of the data division of a COBOL program, is illustrated in the following example:

1. EMPLOYEE-RECORD.
2. EMPLOYEE-NAME.
3. FIRST PICTURE IS X(20).
4. MIDDLE PICTURE IS X(10).
5. LAST PICTURE IS X(20).

02 HOURLY-RATE PICTURE IS 99V99.

The EMPLOYEE-RECORD record consists of the EMPLOYEE-NAME record and the HOURLY-RATE field. The numerals 01, 02, and 05 that begin the lines of the record declaration are **level numbers**, which indicate by their relative values the hierarchical structure of the record

**Implementation of Record Types**

The fields of records are stored in adjacent memory locations. But because the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records. Instead, the offset address, relative to the beginning of the record, is associated with each field. Field accesses are all handled using these offsets. The compile-time descriptor for a record has the general form shown in Figure 6.7. Run-time descriptors for records are unnecessary

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Record |  |
|  |  |  |  |
|  |  | Name |  |
|  |  |  |  |
|  |  | Type |  |
| Field 1 |  |  |
|  |  |  |
|  |  | Offset |  |
|  |  |  |
|  |  |  |  |
|  |  | …... |  |
|  |  |  |  |
|  |  | Name |  |
|  |  |  |  |
|  |  | Type |  |
| field n |  |  |  |
| Offset |  |
|  |  |  |  |
|  |  |  |  |
|  |  | Address |  |
|  |  |  |  |



7)Define an Union Data Type? Explain its implementation In ADA language?

**Union Types**

A **union** is a type whose variables may store different type values at different times during program execution. As an example of the need for a union type, consider a table of constants for a compiler, which is used to store the constants found in a program being compiled. One field of each table entry is for the value of the constant. Suppose that for a particular language being compiled, the types of constants were integer, floating point, and Boolean. In terms of table management, it would be convenient if the same location, a table field, could store a value of any of these three types. Then all constant values could be addressed in the same way. The type of such a location is, in a sense, the union of the three value types it can store.

**Design Issues**

The problem of type checking union types, leads to one major design issue. The other fundamental question is how to syntactically represent a union. In some designs, unions are confined to be parts

of record structures, but in others they are not. So, the primary design issues that are particular to union types are the following:

* Should type checking be required? Note that any such type checking must be dynamic.
* Should unions be embedded in records?

**Ada Union Types**

The Ada design for discriminated unions, which is based on that of its predecessor language, Pascal, allows the user to specify variables of a variant record type that will store only one of the possible type values in the variant. In this way, the user can tell the system when the type checking can be static. Such a restricted variable is called a **constrained variant variable**.

**Implementation of Union Types**

Unions are implemented by simply using the same address for every possible variant. Sufficient storage for the largest variant is allocated. The tag of a discriminated union is stored with the variant in a record like structure. At compile time, the complete description of each variant must be stored. This can be done by associating a case table with the tag entry in the descriptor. The case table has an entry for each variant, which points to a descriptor for that particular variant. To illustrate this arrangement, consider the following Ada example:

**type** Node (Tag : Boolean) **is**

**record**

**case** Tag **is**

**when** True => Count : Integer;

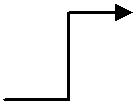
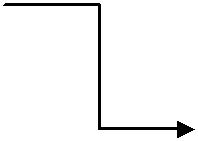
**when** False => Sum : Float;

**end case**;

**end record**;

The descriptor for this type could have the form shown in Figure

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Discriminated union |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| BOOLEAN |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | count |  |  |
| Offset |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | integer |  |  |
|  |  |  |  |  |  |
|  | True | • |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| Address |  |  |  |  |  |  |  |  |
|  | False | • |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | Sum |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | Float |  |  |
|  |  |  |  |  |  |  |  |  |



**Pointer and Reference Types**

* A *pointer* is a variable whose value is an address

– range of values that consists of memory addresses plus a special value, *nil*

* Provide the power of indirect addressing
* Provide a way to manage dynamic memory
* A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)
* Generally represented as a single number

**Pointer Operations:**

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

– Dereferencing can be explicit or implicit

– C++ uses an explicit operation via \*

j = \*ptr

9) Explain pointers in C and C++?

**Pointers in C and C++:**

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

**Pointer Arithmetic in C and C++:**

Float stuff[100];

Float \*p;

p = stuff;

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

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

**Reference Types:**

* **C++** includes a special kind of pointer type called a*reference type*that is used primarily forformal parameters

– Advantages of both pass-by-reference and pass-by-value

* **Java extends C++**’sreference variables and allows them to replace pointers entirely

– References refer to call instances

* **C#** includes both the references of Java and the pointers of C++

**Evaluation of Pointers:**

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

**Introduction to Names and variables:**

* Imperative languages are abstractions of von Neumann architecture
* A machine consists of

– Memory - stores both data and instructions

– Processor - can modify the contents of memory

* Variables are used as an abstraction for memory cells

– For primitive types correspondence is direct

– For structured types (objects, arrays) things are more complicated

–

**9)Define name?explain its design issues?**

**Names:**

* **Why do we need names?**

– need a way to refer to variables, functions, user-defined types, labeled statements, …

* **Design issues for names:**

– Maximum length?

– What characters are allowed?

– Are names case sensitive?

* + - C, C++, and Java names are case sensitive
    - this is not true of other languages

– Are special words reserved words or keywords?

**Length of Names:**

* If too short, they cannot be connotative
* Language examples:

– **FORTRAN** I: maximum 6

– **COBOL**: maximum 30

– **FORTRAN 90 and ANSI C:** maximum 31

– **Ada and Java:** no limit, and all are significant

– **C++:** no limit, but implementers often impose one

**Keywords Vs Reserved Words:**

* Words that have a special meaning in the language
* 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

– most reserved words are also keywords

**10)define variable? Write about its attributes?**

**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

**Variable Attributes:**

* **Name** - not all variables have them
* **Address** -the memory address with which it is associated (l-value)
* **Type** -allowed range of values of variables and the set of defined operations **Value** - the contents of the location with which the variable is associated (r-value

10)Define Binding?

**The Concept of Binding**:

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

– entity could be a variable or a function or even a class

* **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 FORTRAN 77 variable to a memory cell (or a C static variable)
* **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

**Type Binding:**

* **How is a type specified?**

– An ***explicit declaration*** is a program statement used for declaring the types of variables

– An ***implicit declaration*** is a default mechanism for specifying types of variables (the first appearance of the variable in the program)

* **When does the binding take place?**
* If **static,** the type may be specified by either an **explicit or an implicit** declaration

**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

**Compatible type :**

It 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

* A ***type error*** is the application of an operator to an operand of an inappropriate type

•

**When is type checking done?**

* If all type bindings are **static**, nearly all type checking can be **static** (done at compile time)
* If type bindings are **dynamic**, type checking must be **dynamic** (done at run time)
* A programming language is ***strongly typed*** if **type errors** are always detected

– **Advantage**: allows the detection of misuse of variables that result in type errors

**Strong Typing:**

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

(Java is similar)

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

**12)DEFINE EXPRESSION AND EXPLAIN TYPES OF EXPRESSION?**

**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 areevaluated
* 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 +, -

<, >, <=, >=

=, !=

&&

||

**13)DEFINE SHORT CIRCUIT EVALUATION WITH AN EXAMPLE?**

**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

**13) Explain control statements in various programming languages?**

**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
* 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,

**12)explain Unconditional Branching and Guarded Commands?**

•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