LECTURE NOTES

ON

Principles of Programming Languages

III BTTECH IT I SEMESTER

DEPARTMENT OF INFORMATION TECHNOLOGY

CMR TECHNICAL CAMPUS

(Approved by AICTE-NewDelhi, Affiliated to J.N.T.U, Hyderabad)
Kandlakoya(v),Medchal Road,Hyderabad-501 401,Telangana State, India .Website: www.cmrec.ac.in

CONTENTS

UNIT-I

1. Reasons for studying
2. Concepts of programming languages
3. Programming domains
4. Language Evaluation Criteria
5. Influences on Language design
6. Language categories
7. Programming Paradigms – Imperative, Object Oriented, functional Programming, and Logic Programming.
8. Programming Language Implementation – Compilation and Virtual Machines
9. Programming environments.

UNIT-II

1. Introduction
2. primitive
3. character
4. user defined, array
5. Associative, record, union, pointer and reference types, design and implementation uses related to these types.
6. Names, Variable, concept of binding, type checking.
7. Strong typing
8. Type compatibility
9. Named constants
10. Variable initialization

UNIT-III

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

UNIT-IV

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

UNIT-V

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

UNIT - I

PRELIMINARY CONCEPTS

Reasons for Studying Concepts of Programming Languages

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

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

- **Greater ability to learn new languages**
  - Programming languages are still in a state of continuous evolution, which means continuous learning is essential.
  - Programmers who understand the concept of OO programming will have easier time learning Java.
  - Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes easier to see how concepts are incorporated into the design of the language being learned

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

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

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

<u>Programming Domains</u>

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

– Spreadsheets and database systems were developed for business.

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

Language Evaluation Criteria

Readability

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

```
count = count + 1
```

```
count += 1

count ++

++count
```

– Although the last two statements have slightly different meaning from each other and from the others, all four have the same meaning when used as stand-alone expressions.

– Operator overloading where a single operator symbol has more than one meaning.

– Although this is a useful feature, it can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly.

- **Orthogonality**

  – Makes the language easy to learn and read.

  – Meaning is context independent. Pointers should be able to point to any type of variable or data structure. The lack of orthogonality leads to exceptions to the rules of the language.

  – A relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.

  – Every possible combination is legal and meaningful.

  – Ex: page 11 in book.

  – The more orthogonal the design of a language, the fewer exceptions the language rules require.

  – The most orthogonal programming language is ALGOL 68. Every language construct has a type, and there are no restrictions on those types.

  – This form of orthogonality leads to unnecessary complexity.

- **Control Statements**

  – It became widely recognized that indiscriminate use of goto statements severely reduced program readability.

  – Ex: Consider the following nested loops written in C

```
while (incr < 20)

 {
```

```
while (sum <= 100
  {
      sum += incr;
   }
   incr++;
}if C didn't have a loop construct, this would be written as follows:


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

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

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

    timeout = 1 or timeout = true

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

      If used as a variable inside a function, it means the variable is created at compile time.

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

Writability

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

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

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

Reliability

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

Cost

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

– Language implementation system —Free compilers is the key, success of Java‖

– **Reliability**, does the software fail?

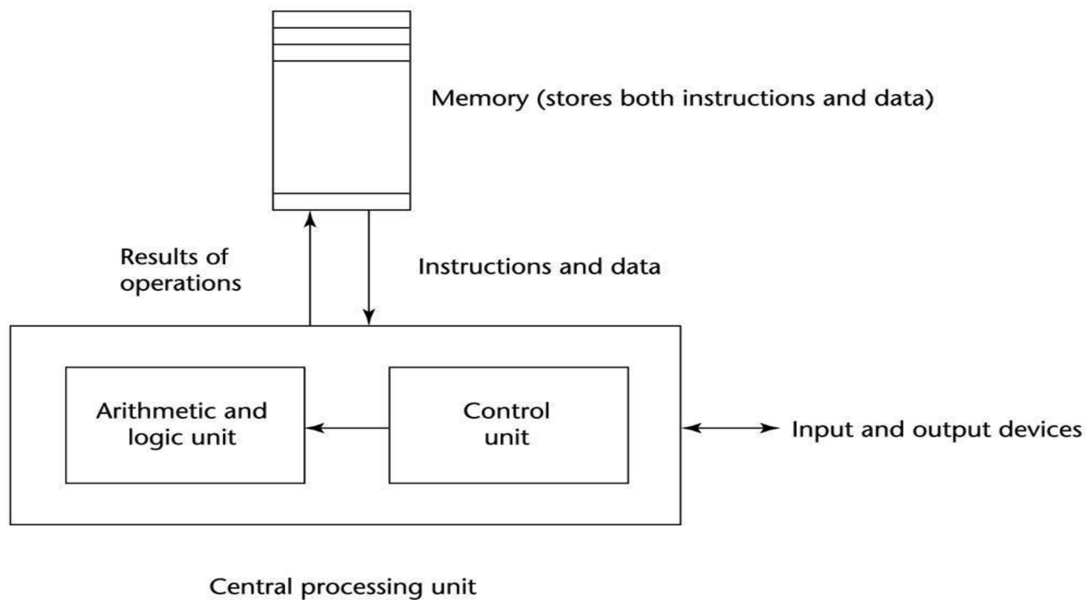
– **Maintaining** programs: Maintenance costs can be as high as two to four times as much as development costs.

– Portability —standardization of the language‖ ,Generality (the applicability to a wide range of applications)

Influences on Language Design

Computer architecture: Von Neumann

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



Programming methodologies

- **1950s and early 1960s:** Simple applications; worry about machine efficiency

- **Late 1960s**: People efficiency became important; readability, better control structures
  - Structured programming
  - Top-down design and step-wise refinement
- **Late 1970s**: Process-oriented to data-oriented
  - data abstraction
- **Middle 1980s**: Object-oriented programming

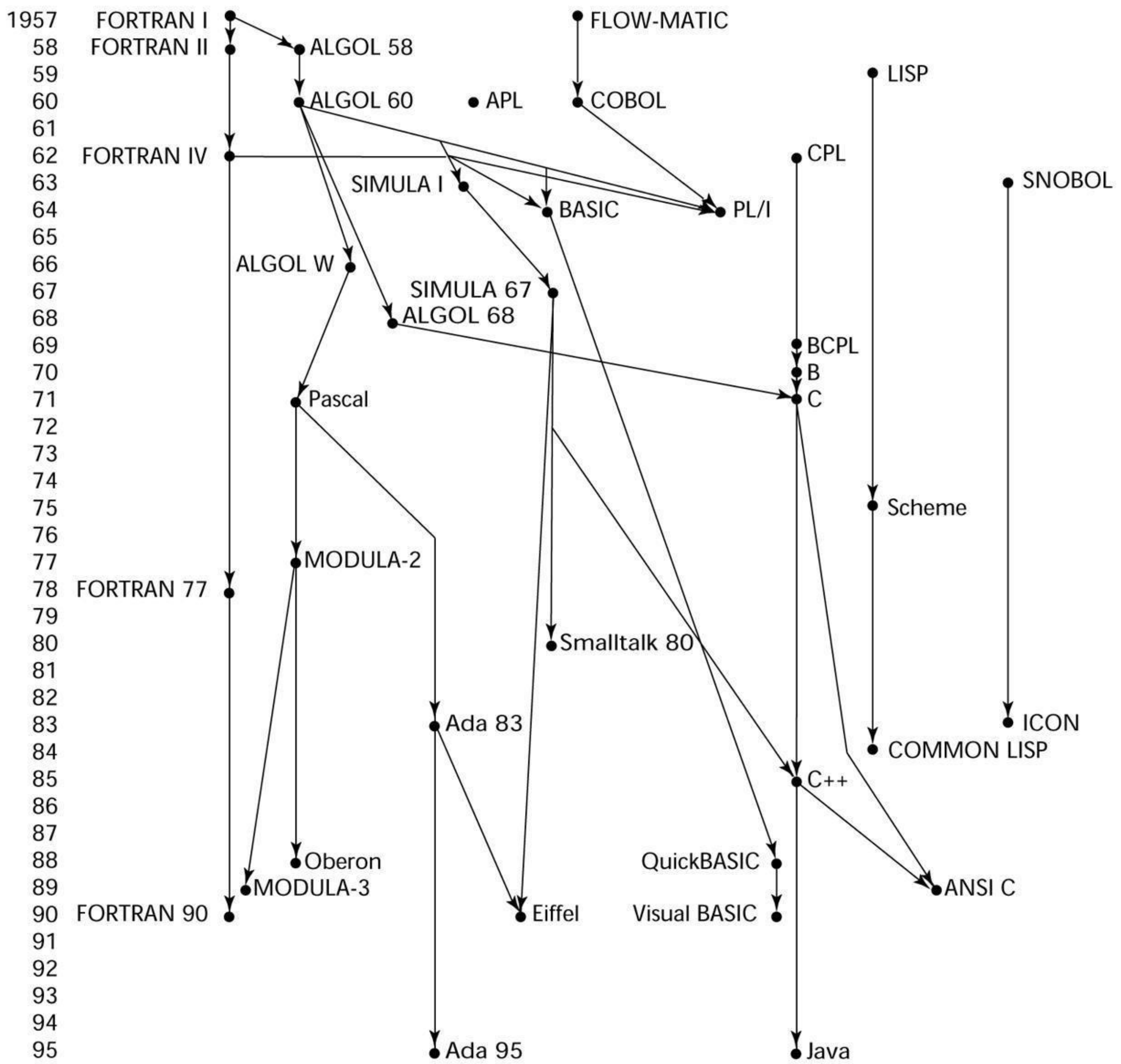
Language Categories

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

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++

- Genealogy of common high-level programming languages



Syntax and Semantics

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

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

Who must use language definitions?

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

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

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 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.

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>

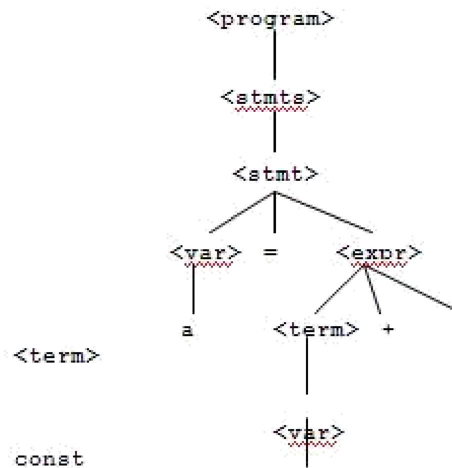
=> a = <term> + <term>

=> 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



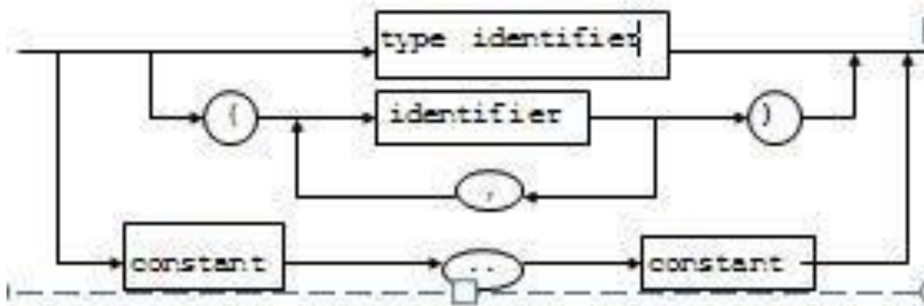
EBNF:

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

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

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

e.g., Pascal type declarations



Static semantics (have nothing to do with meaning)

Categories:

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

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

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

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

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

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

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

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

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

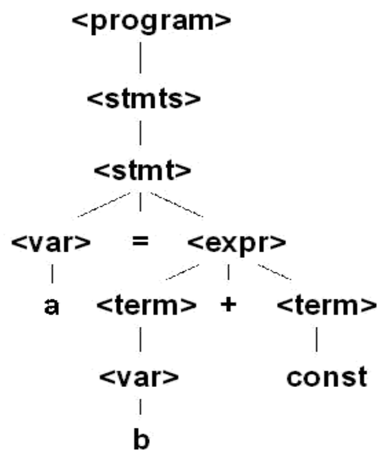
BNF:

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

<var> -> id

PARSE TREE:

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



Ambiguous grammar

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

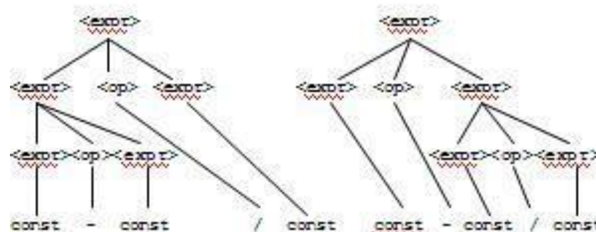
<expr> -> <expr> <op> <expr> | const

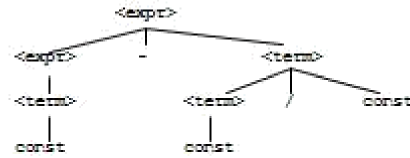
<op> -> / | -

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

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

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





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

=> const - <term>

=> const - <term> / const

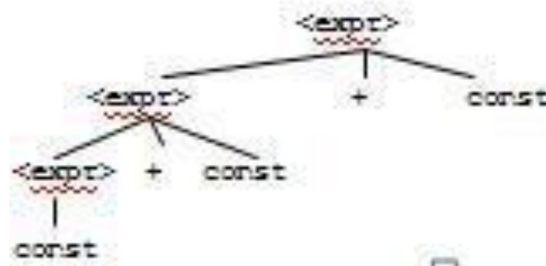
=> const - const / const

Operator associativity can also be indicated by a

grammar

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

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



Extended BNF (just abbreviations):

1. Optional parts are placed in brackets ([]) <proc_call> -> ident [(<expr_list>)]

2. Put alternative parts of RHSs in parentheses and separate them with vertical bars

<term> -> <term> (+ | -) const

3. Put repetitions (0 or more) in braces ({}) <ident> -> letter {letter | digit}

BNF:

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

| <expr> - <term>

| <term>

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

| <term> / <factor>

| <factor>

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

2. Syntax rule: <var> -> id

Semantic rule:

<var>.actual_type <- lookup (id, <var>.env)

How are attribute values computed?

1. If all attributes were inherited, the tree could be decorated in top-down order.
2. If all attributes were synthesized, the tree could be decorated in bottom-up order.
3. 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

4. <expr>.actual_type ← <var>[1].actual_type

<expr>.actual_type =? <expr>.expected_type

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
2. 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***

- An assertion before a statement (a *precondition*) states the relationships and constraints amongvariables that are true at that point in execution

- An assertion following a statement is a *postcondition*

- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

- Pre-post form: {P} statement {Q}

- *An example:* a := b + 1 {a > 1}

  One possible precondition: {b > 10}

  Weakest precondition: {b > 0}

Program proof process: The postcondition for the whole program is the desired results. Work back through the program to the first statement. If the precondition on the first statement is the same as the program spec, the program is correct.

An axiom for assignment statements:

An axiom for assignment statements

$\{Q_{x \to E}\}$ x := E {Q}

The Rule of Consequence:

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

An inference rule for sequences

- For a sequence S1;S2:

  {P1} S1 {P2}
  {P2} S2 {P3}

  the inference rule is:

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

An inference rule for logical pretest loops

For the loop construct:

{P} while B do S end {Q}

the inference rule is:

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\} \text{ while } B \text{ do } S\ \{I \text{ and } (\text{not } B)\}}$$

where I is the loop invariant. Characteristics of the loop invariant

I must meet the following conditions:

1. P => I (the loop invariant must be true initially)
2. {I} B {I} (evaluation of the Boolean must not change the validity of I)
3. {I and B} S {I} (I is not changed by executing the body of the loop)
4. (I and (not B)) => Q (if I is true and B is false, Q is implied)
5. The loop terminates (this can be difficult to prove)

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.

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

- **Evaluation of axiomatic semantics**:

1. Developing axioms or inference rules for all of the statements in a language is difficult
2. It is a good tool for correctness proofs, and excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers

Denotational Semantics

- 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

$$s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$$

- 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)

M_{dec} ('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1‘) = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

2. Expressions

M_e(<expr>, s) Δ=

case <expr> of

<dec_num> => M_{dec}(<dec_num>,

s) <var> =>

if VARMAP(<var>, s) =

undef then error

else VARMAP(<var>, s)

<binary_expr> =>

if (M_e(<binary_expr>.<left_expr>, s) = undef

OR M_e(<binary_expr>.<right_expr>, s) =

undef)

then error

else

if (<binary_expr>.<operator> = ë+í then

M_e(<binary_expr>.<left_expr>, s) +

M_e(<binary_expr>.<right_expr>, s)

else M_e(<binary_expr>.<left_expr>, s) *

M_e(<binary_expr>.<right_expr>, s)

3 Assignment Statements

M_a(x := E, s) Δ=

if M_e(E, s) = error

then error

else $s' = \{<i_1',v_1'>,<i_2',v_2'>,...,<i_n',v_n'>\}$,

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

v_j' = VARMAP(i_j, s) if i_j <> x

= M_e (E, s) if i_j = x

4 Logical Pretest Loops

M_l(while B do L, s) Δ=

if M_b(B, s) =

undef then error

else if M_b(B, s) =

false then s

else if M_{sl} (L, s) = error

then error

else M_l(while B do L, M_{sl}(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

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

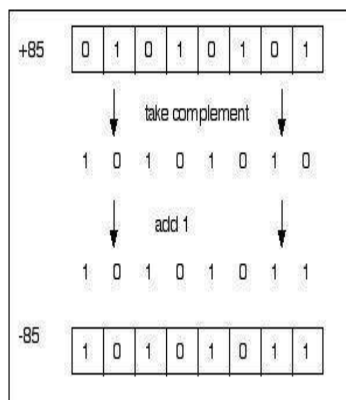
– Representing negative integers.

Representing negative integers:

- Sign bit
- Ones complement

Twos Complement:

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



<u>Floating Point Types</u>:

- 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

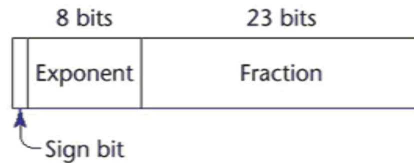
- The collection of values that can be represented by a floating-point type is defined in terms of precision and range
- **Precision** is the accuracy of the fractional part of a value, measured as the number of bits
- **Range** is a combination of the range of fractions and, more important, the range of exponents.
- Usually based on hardware
- IEEE Floating-Point Standard 754
  - –32 and 64 bit standards

Representing Real Numbers:

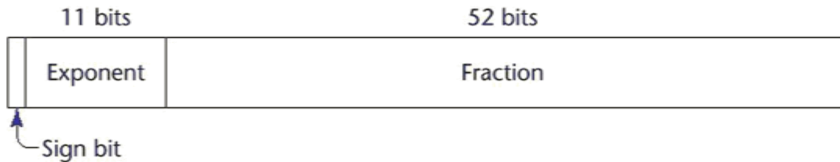
- We can convert the decimal number to base 2 just as we did for integers
- How do we represent the decimal point?
  - – fixed number of bits for the whole and fractional parts severely limits the range of values we can represent
- Use a representation similar to scientific notation

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 $2^{Exponent}$



(a)



(b)

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:** 1.0×10^{-28} to 7.9×10^{28}

- **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.

- **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

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

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)

Associative Arrays

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"};
```

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.
```

5 FIRST PICTURE IS X(20).

5 MIDDLE PICTURE IS X(10).

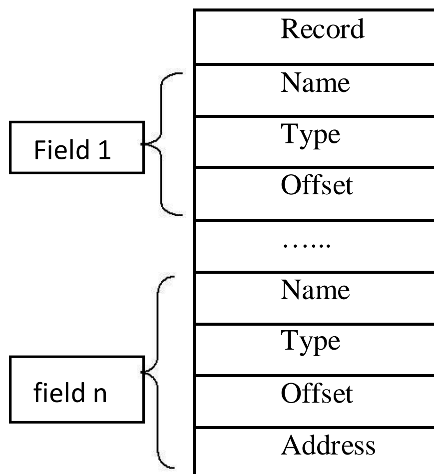
5 LAST PICTURE IS X(20).

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



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**

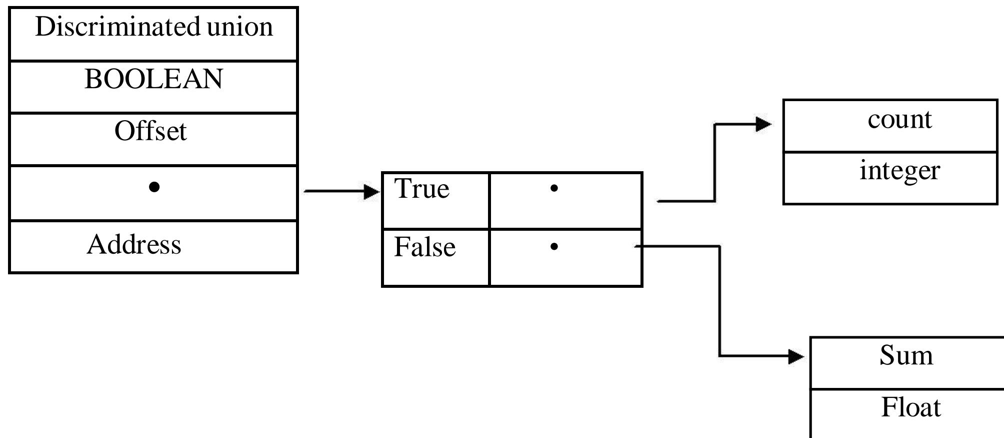
when True => Count : Integer;

when False => Sum : Float;

end case;

end record;

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



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 *

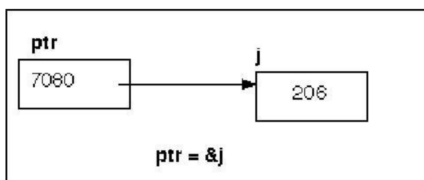
j = *ptr

Pointer Operations Illustrated:

Pointer Operations Illustrated

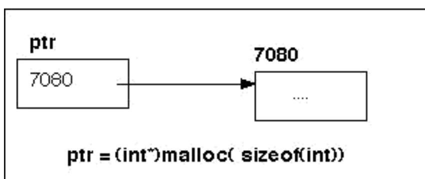
assignment

ptr = &j



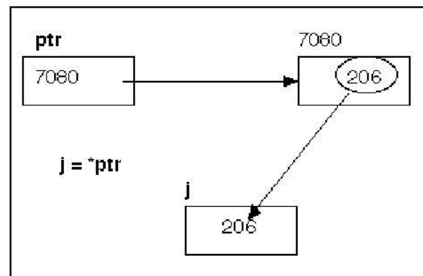
allocation

ptr = (int*)malloc(



Dereferencing a pointer

j = *ptr



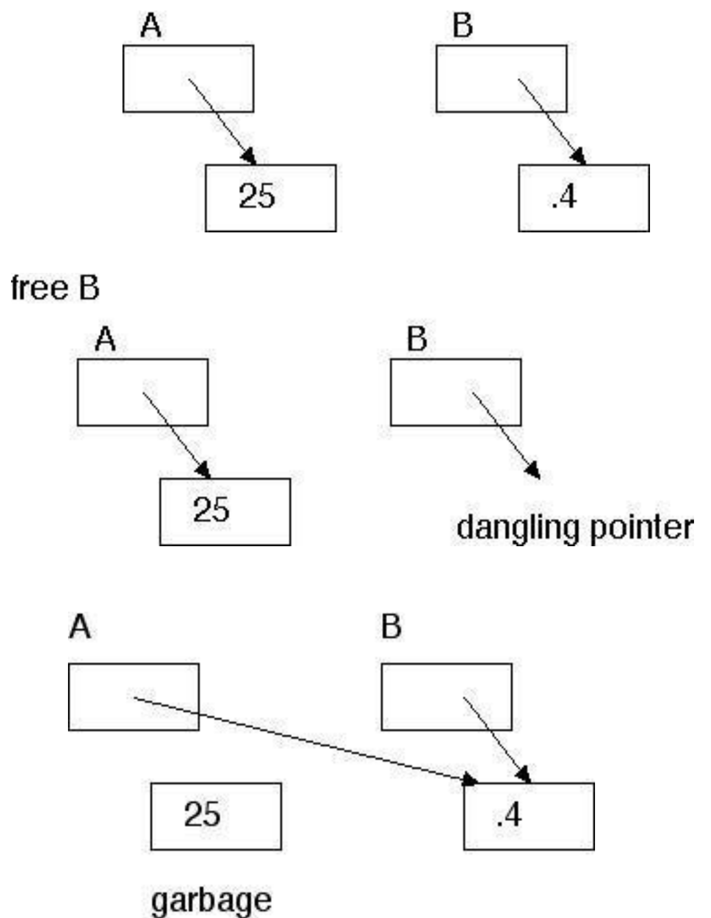
Pointer Problems:

Pointer

Problems

- Dangling pointers (dangerous)
  - A pointer points to a heap – dynamic variable that has been de-allocated
- Garbage
  - An allocated heap-dynamic variable that is

Addison-Wes ey



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 for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- **Java extends C++**'s reference variables and allows them to replace pointers entirely
  - References 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
  -

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

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

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

How strongly typed?

– **FORTRAN 77 is not**: parameters, EQUIVALENCE

– **Pasca**l has variant records

– **C and C++**

  - parameter type checking can be avoided
  - unions are not type checked

– **Ada** is almost

  - **UNCHECKED CONVERSION** is loophole

– **Java** is similar

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

Type Compatibility

- Our concern is primarily for structured types
- Def: Name type compatibility means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name

- Easy to implement but highly restrictive:
- Subranges of integer types are not compatible with integer types
- Formal parameters must be the same type as their corresponding actual parameters (Pascal)
- Structure type compatibility means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
- Consider the problem of two structured types:

  Are two record types compatible if they are structurally the same but use different field names?
- Are two array types compatible if they are the same except that the subscripts are different?

  (e.g. [1..10] and [0..9])
- Are two enumeration types compatible if their components are spelled differently?
- With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Named Constants

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

Variable Initialization

- Def: The binding of a variable to a value at the time it is bound to storage is called initialization
- Initialization is often done on the declaration statement

e.g., Java

int sum = 0;

Coercion:

- The automatic conversion between types is called **coercion**.
- **Coercion rules** they can weaken typing considerably
  - **C and C++** allow both **widening and narrowing coercions**
  - **Java** allows only **widening coercions**
    - **Java's strong typing** is still far less effective than that of **Ada**

Dynamic Type Binding:

- Dynamic Type Binding (**Perl, JavaScript and PHP, Scheme**)
- Specified through an assignment statement e.g**., JavaScript**

  List = [2, 4.33, 6, 8];

  List = 17.3;

- This provides a lot of flexibility
- But …
  - How do you do any type checking?

Storage Bindings & Lifetime:

- The lifetime of a variable is the time during which it is bound to a particular memory cell
  - **Allocation -** getting a cell from some pool of available cells
  - **Deallocation** - putting a cell back into the pool
- Depending on the language, allocation can be either controlled by the programmer or done automatically

Categories of Variables by Lifetimes:

- **Static--**lifetime is same as that of the program
- **Stack-dynamic**--lifetime is duration of subprogram

- ***Explicit heap-dynamic*** -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- ***Implicit heap-dynamic***--Allocation and deallocation caused by assignment statements

Variable 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
- Scope and lifetime are sometimes closely related, but are different concepts

Static Scope:

- The **scope of a variable** can be determined from the **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

Scope and Shadowed Variables:

- Variables can be hidden from a unit by having a "**closer**" variable with the same name
- **C++, Java and Ada** allow access to some of these "hidden" variables
  - **In Ada**: **unit.name**
  - **In C++: class_name::name** or **::name** for globals
  - **In Java**: **this.name** for variables declared at class level

Static Scoping and Nested Functions:

- Assume MAIN calls A and B

A calls C and D

B calls A and E

Nested Functions and Maintainability:

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

Dynamic Scope:

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point
- This is easy to implement but it makes programs hard to follow
- Used in APL, SNOBOL, early versions of LISP

– Perl allows you to use dynamic scope for selected variablesScope

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

Referencing Environments:

- The *referencing environment* of a statement is the collection of all names that are visible in the statement

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

  – In a **dynamic-scoped language**, the referencing environment is the local variables plus all visible variables in all active subprograms subprogram is active if its execution has begun but has not yet terminate

Expressions and Statements

Arithmetic Expressions:

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

Issues for Arithmetic Expressions:

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

Operators:

A unary operator has one operand

- unary -, !

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

Conditional Expressions:

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

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

- Evaluates as if written like **if**

  (count == 0) average = 0

  else average = sum /count

Operator Precedence Rules:

- *Precedence rules* define the order in which ―adjacent‖ operators of different precedence levels are evaluated
- Typical precedence levels

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

Operator Associativity Rules:

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

Operand Evaluation Order:

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

Potentials for Side Effects:

- *Functional side effects:* when a function changes a two-way parameter or a non-local variable
- Problem with functional side effects:
  - When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

    a = 10;

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

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

Functional Side Effects:

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

Overloaded Operators:

- Use of an operator for more than one purpose is called *operator overloading*
- Some are common (e.g., + for int and float)
- Some are potential trouble (e.g., * in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability
  - Can be avoided by introduction of new symbols (e.g., Pascal‘s div for integer division)

Type Conversions

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

Coercion:

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

Casting:

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

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

Errors in Expressions:

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

Relational Operators:

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

Boolean Operators:

- Operands are Boolean and the result is Boolean

- **Example o**perators

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

No Boolean Type in C:

- C has no Boolean type
  - it uses int type with 0 for false and nonzero for true
- **Consequence**

  –a < b < c is a legal expression

  - the result is not what you might expect:
    - Left operator is evaluated, producing 0 or 1
    - This result is then compared with the third operand (i.e., c)

Precedence of C-based operators:

postfix ++, --

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

*,/,%

binary +, -

<, >, <=, >=

=, !=

&&

||

Short Circuit Evaluation:

- Result is determined without evaluating all of the operands and/or operators

- Example: (13*a) * (b/13–1)

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

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

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

Index++;

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

Mixed-Mode Assignment:

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

int a, b;

float c;

c = a / b;

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

Assignment Statements:

- The general syntax

<target_var> <assign_operator> <expression>

- The assignment operator

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

:= **ALGOLs, Pascal, Ada**

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

Compound Assignment:

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

a = a + b

is written as

a += b

Unary Assignment Operators:

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
  - These have side effects
- Examples

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

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

Count++ (count incremented)

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

Assignment as an Expression:

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

While ((ch = get char ())!= EOF){…}

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

Control Statements: Evolution

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

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

Control Structure

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

execution it controls

- Design question

–Should a control structure have multiple entries?

Selection Statements

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:

–Two-way selectors

–Multiple-way selectors

Two-Way Selection Statements

- General form:

If control_expression

then clause

else clause

- **Design Issues**:

–What is the form and type of the control expression?

–How are the **then** and **else** clauses specified?

–How should the meaning of nested selectors be specified?

The Control Expression

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

Clause Form

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

```
if x > y :
x = y
print "case 1"
```

Nesting Selectors

- **Java example**

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

- Which if gets the else?
- Java's static semantics rule: else matches with the nearest

if Nesting Selectors (continued)

- To force an alternative semantics, compound statements may be used:

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

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

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

result = 1

end

end

•Python

if sum == 0 :

if count == 0 :

result = 0

else :

result = 1

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups

Design Issues:

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

Multiple-Way Selection: Examples

- C, C++, and Java

switch (expression) { case

const_expr_1: stmt_1;

…

case const_expr_n: stmt_n;

[default: stmt_n+1]

}

- Design choices for C_s **switch** statement
- Control expression can be only an integer type
- Selectable segments can be statement sequences, blocks, or compound statements
- Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
- **default** clause is for unrepresented values (if there is no **default**, the whole statement does nothing)

Multiple-Way Selection: Examples

- **C#**
  - Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (goto or break)
- **Ada**

case expression is

when choice list => stmt_sequence;

…

when choice list => stmt_sequence;

when others => stmt_sequence;]

end case;

- More reliable than C_s switch (once a stmt_sequence execution is completed, control is passed to the first statement after the case statement
- **Ada design choices**:

1. Expression can be any ordinal type
2. Segments can be single or compound
3. Only one segment can be executed per execution of the construct
4. Unrepresented values are not allowed

- **Constant List Forms**:

1. A list of constants
2. Can include:

- **Subranges**
- Boolean OR operators (|)

Multiple-Way Selection Using if

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

example in Python:

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

bag3 = True

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:

1. How is iteration controlled?
2. Where is the control mechanism in the loop?

Counter-Controlled Loops

- A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values
- **Design Issues**:
- What are the type and scope of the loop variable?
- What is the value of the loop variable at loop termination?
- Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?

Iterative Statements: Examples

- FORTRAN 95 syntax

- Stepsize can be any value but zero
- Parameters can be expressions
- **Design choices**:

1. Loop variable must be **INTEGER**

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

•FORTRAN 95 : a second form:

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

…

End Do [name]

- Cannot branch into either of Fortran_s Do statements

• **Ada**

for var in [reverse] discrete_range loop ...

end loop

• **Design choices**:

- Type of the loop variable is that of the discrete range (A discrete range is a sub-range of an integer or enumeration type).
- Loop variable does not exist outside the loop
- The loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control
- The discrete range is evaluated just once

• Cannot branch into the loop body

• C-based languages

for ([expr_1] ; [expr_2] ; [expr_3]) statement

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas

–The value of a multiple-statement expression is the value of the last statement in the expression

–If the second expression is absent, it is an infinite loop •**Design choices:**

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

•C++ differs from C in two ways:

•The control expression can also be Boolean

•The initial expression can include variable definitions (scope is from the definition to the end of the loop body)

•Java and C#

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

Iterative Statements: Logically-Controlled Loops

•Repetition control is based on a Boolean expression

•Design issues:

–Pretest or posttest?

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

Iterative Statements: Logically-Controlled Loops: Examples

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

while (ctrl_expr) do

loop body loop body

while (ctrl_expr)

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

Iterative Statements: Logically-Controlled Loops: Examples

•Ada has a pretest version, but no posttest

•FORTRAN 95 has neither

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

Iterative Statements: User-Located Loop Control Mechanisms

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

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

•Design issues for nested loops

•Should the conditional be part of the exit?

•Should control be transferable out of more than one loop?

Iterative Statements: User-Located Loop Control Mechanisms break and continue

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

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

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

•Java and Perl have labeled versions of **continue Iterative Statements**: Iteration Based on Data Structures •Number of elements of in a data structure control loop iteration

•Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate

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

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

•C#_s **foreach** statement iterates on the elements of arrays and other collections:

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

- The notation {0} indicates the position in the string to be displayed •Perl has a built-in iterator for arrays and hashes, **foreach**

Unconditional Branching

•Transfers execution control to a specified place in the program

•Represented one of the most heated debates in 1960_s and 1970_s

•Well-known mechanism: goto statement

•Major concern: Readability

•Some languages do not support goto statement (e.g., Java)

•C# offers goto statement (can be used in switch statements)

•Loop exit statements are restricted and somewhat camouflaged goto_s

Guarded Commands

•Designed by Dijkstra

•Purpose: to support a new programming methodology that supported verification (correctness) during development

•Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada)

• **Basic Idea:** if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

•Form

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

[] <Boolean exp> -> <statement>

fi

•Semantics: when construct is reached,

–Evaluate all Boolean expressions

–If more than one are true, choose one non-deterministically

–If none are true, it is a runtime error

Selection Guarded Command: Illustrated

Loop Guarded Command

- **Form**

do <Boolean> -> <statement>

[] <Boolean> -> <statement>

...

[] <Boolean> -> <statement>

od

•Semantics: for each iteration

–Evaluate all Boolean expressions

–If more than one are true, choose one non-deterministically; then start loop again

–If none are true, exit loop

Guarded Commands: Rationale

- Connection between control statements and program verification is intimate

•Verification is impossible with goto statements

•Verification is possible with only selection and logical pretest loops

•Verification is relatively simple with only guarded commands

UNIT-III

SUB PROGRAMS AND BLOCKS

Basic Definitions:

- A subprogram definition is a description of the actions of the subprogram abstraction
- A subprogram call is an explicit request that the subprogram be executed
- A subprogram header is the first line of the definition, including the name, the kind of
- subprogram, and the formal parameters
- The parameter profile of a subprogram is the number, order, and types of its parameters
- The protocol of a subprogram is its parameter profile plus, if it is a function, its return type
- A subprogram declaration provides the protocol, but not the body, of the subprogram
- A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram
- An actual parameter represents a value or address used in the subprogram call

Statement

Actual/Formal Param Correspondence

<u>Two basic choices:</u>

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

Advantage: order is irrelevant

Disadvantage: user must know the formal

- parameter‘s names

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

For named association:

Advantage: order is irrelevant

Disadvantage: user must know the formal parameter's names

Default Parameter Values

Example, in Ada:

procedure sort (list : List_Type;

length : Integer := 100);

-----

sort (list => A);

Two Types of Subprograms

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

Design Issues for Subprograms

- What parameter passing methods are provided?
- Are parameter types checked?
- Are local variables static or dynamic?
- What is the referencing environment of a passed subprogram?
- Are parameter types in passed subprograms checked?
- Can subprogram definitions be nested?
- Can subprograms be overloaded?
- Are subprograms allowed to be generic?
- Is separate/independent compilation supported?
- Referencing Environments
- If local variables are stack-dynamic:
- **Advantages:**
- Support for recursion
- Storage for locals is shared among some subprograms
- **Disadvantages:**

- Allocation/deallocation time
- Indirect addressing
- Subprograms cannot be history sensitive
- Static locals are the opposite

Parameters and Parameter Passing:

Semantic Models: in mode, out mode, inout mode

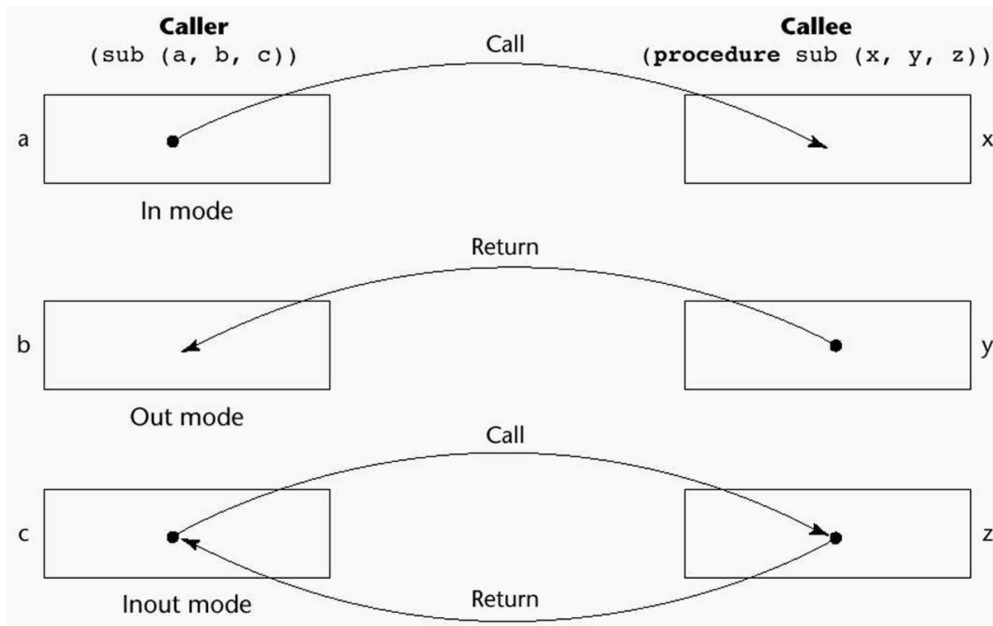
Conceptual Models of Transfer:

- Physically move a value
- Move an access path

Implementation Models:

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

Models of Parameter Passing



Pass-By-Value

- in mode
- Either by physical move or access path

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

Disadvantages of physical move:

Requires more storage

Cost of the moves

Pass-By-Result

- out mode

Local's value is passed back to the caller

- Physical move is usually used
-

Disadvantages:

- If value is moved, time and space

- In both cases, order dependence may be a problem

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

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

Pass-By-Value-Result

inout mode

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

Disadvantages:

Those of pass-by-result

Those of pass-by-value

Pass-By-Reference

inout mode

Pass an access path Also called pass-by-sharing

Advantage: passing process is efficient

Disadvantages:

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

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

Pass-By-Name multiple modes By textual substitution ,Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment

Purpose: flexibility of late binding

Resulting semantics:

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

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

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

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

Pass-By-Name Example 1

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

begin

x := 1;

y := 2;

x := 2;

y := 3;

end;

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

Pass-By-Name Example 2

- Assume k is a global variable

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

begin

k := 1;

y := x;
```

```
k := 5;

z := x;

end;

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

Disadvantages of Pass-By-Name

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

Param Passing: Language Examples

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

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

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

Param Passing: PL Example

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

Ada

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

Type Checking Parameters

Now considered very important for reliability

FORTRAN 77 and original C: none

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

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

Implementing Parameter Passing

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

Name:

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

Ada Param Passing Implementations

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

Multidimensional Arrays as Params

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

C and C++

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

More Array Passing Designs

Pascal Not a problem (declared size is part of the array's type)

Ada

Constrained arrays—like Pascal

Unconstrained arrays—declared size is part of the object declaration Pre-90 FORTRAN , Formal parameter declarations for arrays can include passed parameters

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

 INTEGER ROWS, COLS

 REAL MATRIX (ROWS, COLS), RESULT

  ...

  END
```

Design Considerations for Parameter Passing

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

Subprograms As Parameters: Issues

Are parameter types checked?

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

What is the correct referencing?

environment for a subprogram that was sent as a parameter?

Possibilities:

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

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

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

For static-scoped languages, deep binding is most natural

For dynamic-scoped languages, shallow binding is most natural

Overloading

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

Generic Subprograms

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

Language Examples:

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

Functions

Design Issues:

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

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

Accessing Nonlocal Environments

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

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

Methods for Accessing Non locals

FORTRAN COMMON

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

Can be used to share data or share storage

Static scoping

External declarations: C

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

User-Defined Overloaded Operators

Nearly all programming languages have overloaded operators

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

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

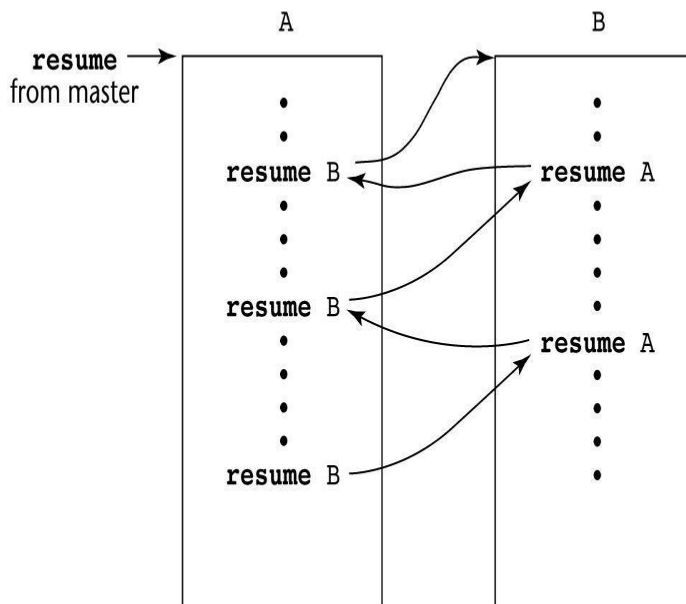
function "*"(a, b : inVector_Type) return Integer is

```
sum : Integer := 0;

begin

for index in a _range loop

sum := sum + a(index) * b(index);

end loop;

return sum;

end "*";
```

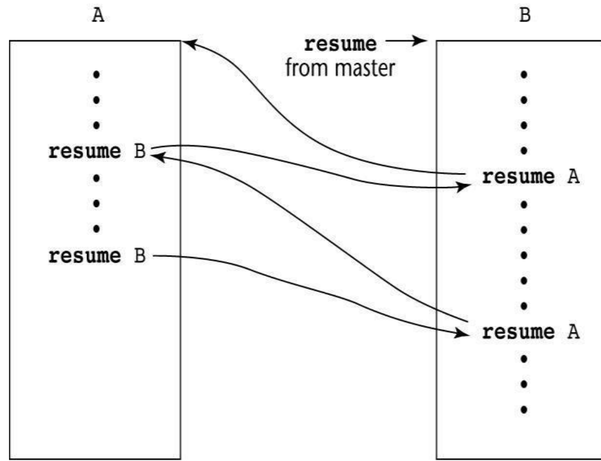
Are user-defined overloaded operators good or bad?

Coroutines:

Coroutine is a subprogram that has multiple entries and controls them itself Also called symmetric control A coroutine call is named a resume. The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine. Typically, coroutines repeatedly resume each other, possibly forever. Coroutines provide quasiconcurrent execution of program units (the coroutines) Their execution is interleaved, but not overlapped



Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops

UNIT-IV

ABSTRACT DATA TYPES

Abstraction:

- The concept of abstraction is fundamental in programming

- Nearly all programming languages support process abstraction with subprograms

- Nearly all programming languages designed since 1980 have supported data abstraction with some kind of module

Encapsulation:

- **Original motivation**:

  - Large programs have two special needs:

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

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

  - **Obvious solution**: a grouping of subprograms that are logically related into a unit that can be separately compiled

      - These are called encapsulations

Examples of Encapsulation Mechanisms:

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

2. FORTRAN 77 and C - Files containing one or more subprograms can be independently compiled

3. FORTRAN 90, C++, Ada (and other contemporary languages) - separately compliable modules

Definitions: An abstract data type is a user-defined datatype that satisfies the following two conditions:

Definition 1: The representation of and operations of objects of the type are defined in a single syntactic unit; also, other units can create objects of the type.

Definition 2: The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition.

Advantage of Restriction 1:

- Same as those for encapsulation: program organization, modifiability (everything associated with a data structure is together), and separate compilation

Advantage of Restriction 2:

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

Built-in types are abstract data types:

EX: int type in C

- The representation is hidden
- Operations are all built-in
- User programs can define objects of int type
- User-defined abstract data types must have the same characteristics as built-in abstract data types

Language Requirements for Data Abstraction:

1. A syntactic unit in which to encapsulate the type definition.

2. A method of making type names and subprogram headers visible to clients, while hiding actual definitions.

3. Some primitive operations must be built into the language processor (usually just assignment and comparisons for equality and inequality)

- Some operations are commonly needed, but must be defined by the type designer

- **e.g.**, iterators, constructors, destructors

Language Design Issues:

1. Encapsulate a single type, or something more?

2. What types can be abstract?

3. Can abstract types be parameterized?

4. What access controls are provided?

Language Examples:

1. Simula 67

- Provided encapsulation, but no information hiding

2. Ada

- The encapsulation construct is the package

- **Packages usually have two parts:**

1. Specification package (the interface)

2. Body package (implementation of the entities named in the specification

- Any type can be exported

- Information Hiding

- Hidden types are named in the spec package in, as in:

type NODE_TYPE is private;

- Representation of an exported hidden type is specified in a special invisible (to clients) part of the spec package (the private clause), as in:

```
package … is

type NODE_TYPE is private;

…

type NODE_TYPE is

  record

  …

  end record;

…
```

- A spec package can also define unhidden type simply by providing the representation outside a private clause

The reasons for the two-part type definition are:

1. The compiler must be able to see the representation after seeing only the spec package (the compiler can see the private clause)

2. Clients must see the type name, but not the representation (clients cannot see the private clause)

Parameterized Abstract Data Types

- Parameterized ADTs allow designing an ADT that can store any type elements (among other things)

- Also known as generic classes

- C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs

Parameterized ADTs in Ada

- Ada Generic Packages

–Make the stack type more flexible by making the element type and the size of the stack generic

```
generic
Max_Size: Positive;
type Elem_Type is private;
package Generic_Stack is
Type Stack_Type is limited private;
function Top(Stk: in out StackType) return Elem_type;
…
end Generic_Stack;
Package Integer_Stack is new Generic_Stack(100,Integer);
Package Float_Stack is new Generic_Stack(100,Float);
```

Parameterized ADTs in C++

- Classes can be somewhat generic by writing parameterized constructor functions

```
class stack {
…
stack (int size)  {
stk_ptr = new int [size];
max_len = size - 1;
top = -1;
```

```
};
…
}
stack stk(100);
```

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

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

stack() {
stackPtr = new Type[100];
maxLen = 99;
topPtr = -1;
}
…
}
```

Parameterized Classes in Java 5.0

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

Parameterized Classes in C# 2005

- Similar to those of Java 5.0
- Elements of parameterized structures can be accessed through indexing
- The concept of ADTs and their use in program design was a milestone in the development of languages
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding
- Ada provides packages that simulate ADTs
- C++ data abstraction is provided by classes
- java_s data abstraction is similar to C++

- Ada, C++, Java 5.0, and C# 2005 support parameterized ADTs

Object-Oriented Programming

- Abstract data types
- Inheritance

–Inheritance is the central theme in OOP and languages that support it

- Polymorphism

Inheritance

- Productivity increases can come from reuse

–ADTs are difficult to reuse—always need changes

–All ADTs are independent and at the same level

- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
- Inheritance addresses both of the above concerns--reuse ADTs after minor changes and define classes in a hierarchy

Object-Oriented Concepts

- ADTs are usually called *classes*
- Class instances are called *objects*
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *super class*
- Subprograms that define operations on objects are called *methods*
- Calls to methods are called *messages*
- The entire collection of methods of an object is called its *message protocol* or *message interface*
- Messages have two parts--a method name and the destination object
- In the simplest case, a class inherits all of the entities of its parent

- Inheritance can be complicated by access controls to encapsulated entities
  –A class can hide entities from its subclasses
  –A class can hide entities from its clients
  –A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
  –The new one *overrides* the inherited one
  –The method in the parent is *overridden*
- There are two kinds of variables in a class:

***Class variables* -** one/class

Instance variables - one/object

- There are two kinds of methods in a class:

Class methods – accept messages to the class

Instance methods – accept messages to objects

Dynamic Binding

- A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants
- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic
- Allows software systems to be more easily extended during both development and maintenance

Dynamic Binding Concepts

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- An *abstract class* is one that includes at least one virtual method
- An abstract class cannot be instantiated

Design Issues for OOP Languages: The Exclusivity of Objects

•Are Subclasses Subtypes

•Type Checking and Polymorphism

•Single and Multiple Inheritance

•Object Allocation and DeAllocation

•Dynamic and Static Binding

•Nested Classes

Concurrency can occur at four levels:

1. Machine instruction level

2. High-level language statement level

3. Unit level

4. Program level

Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

The Evolution of Multiprocessor Architectures:

1. Late 1950s - One general-purpose processor and one or more special-purpose processors for input and output operations

2. Early 1960s - Multiple complete processors, used for program-level concurrency

3. Mid-1960s - Multiple partial processors, used for instruction-level concurrency

4. **Single-Instruction Multiple-Data (SIMD) machine** The same instruction goes to all processors, each with different data - e.g., *vector processors*

5. Multiple-Instruction Multiple-Data (MIMD) machines, Independent processors that can be synchronized (unit-level concurrency)

Def: A thread of control: in a program is the sequence of program points reached as control flows through the program

Categories of Concurrency:

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

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

- Coroutines *provide only* **quasiconcurrency**

Reasons to Study Concurrency:

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

2. Computers capable of physical concurrency are now widely used

Fundamentals (for stmt-level concurrency) :

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

- Tasks differ from ordinary subprograms in that:

1. A task may be implicitly started
2. When a program unit starts the execution of a task, it is not necessarily suspended
3. When a task's execution is completed, control may not return to the caller

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

- **Task communication can be through:**

1. Shared nonlocal variables

2. Parameters

3. Message passing

- **Kinds of synchronization**:

1. Cooperation

- Task A must wait for task B to complete some specific activity before task A can continue its execution

e.g., the producer-consumer problem

2. Competition

- When two or more tasks must use some resource that cannot be simultaneously used

e.g., a shared counter

- A problem because operations are not atomic

- Competition is usually provided by **mutually exclusive access** (methods are discussed later

- Providing synchronization requires a mechanism for delaying task execution

- Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors

- Tasks can be in one of several different execution states:

1. New - created but not yet started
2. Runnable or ready - ready to run but not currently running (no available processor)
3. Running
4. Blocked - has been running, but cannot not continue (usually waiting for some event to occur)
5. Dead - no longer active in any sense

- In sequential code, it means the unit will eventually complete its execution
- In a concurrent environment, a task can easily lose its liveness
- If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

- **Design Issues for Concurrency**:

1. How is cooperation synchronization provided?
2. How is competition synchronization provided?
3. How and when do tasks begin and end execution?
4. Are tasks statically or dynamically created?

Example: A buffer and some producers and some consumers

Technique: Attach two SIGNAL objects to the buffer, one for full spots and one for empty spots

Methods of Providing Synchronization:

1. Semaphores
2. Monitors
3. Message Passing

1. Semaphores (Dijkstra - 1965)

- A **semaphore** is a data structure consisting of a counter and a queue for storing task descriptors
- Semaphores can be used to implement guards on the code that accesses shared data structures
- Semaphores have only two operations, wait and release (originally called P and V by Dijkstra)
- Semaphores can be used to provide both competition and cooperation synchronization

Cooperation Synchronization with Semaphores:

***Example:* A shared buffer**

- The buffer is implemented as an ADT with the operations **DEPOSIT and FETCH** as the only ways to access the buffer.

Use two semaphores for cooperation:

Empty spots and full spots

- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer
- DEPOSIT must first check empty spots to see if there is room in the buffer
- If there is room, the counter of empty spots is decremented and the value is inserted
- If there is no room, the caller is stored in the queue of empty spots
- When DEPOSIT is finished, it must increment the counter of full spots
- FETCH must first check full spots to see if there is a value
- If there is a full spot, the counter of full spots is decremented and the value is removed
- If there are no values in the buffer, the caller must be placed in the queue of full spots
- When FETCH is finished, it increments the counter of empty spots
- The operations of FETCH and DEPOSIT on the semaphores are accomplished through two

semaphore operations named wait and release wait (aSemaphore)

```
if a Semaphore's counter > 0 then  Decrement aSemaphore's counter
else
  Put the caller in aSemaphore's queue Attempt to transfer control to some
   ready task
  (If the task ready queue is empty, deadlock occurs)
end
release(aSemaphore)
  if aSemaphore's queue is empty then
  Increment aSemaphore's counter
  else
    Put the calling task in the task ready queue Transfer control to a task from aSemaphore's queue
```

end

- **Competition Synchronization with Semaphores:**

  - A third semaphore, named access, is used to control access (competition synchronization)
  - The counter of access will only have the values 0 and 1
  - Such a semaphore is called a **binary semaphore**

SHOW the complete shared buffer example

- Note that wait and release must be atomic!

Evaluation of Semaphores:

1. Misuse of semaphores can cause failures in cooperation synchronization
   e.g., the buffer will overflow if the wait of full spots is left out
2. Misuse of semaphores can cause failures in competition synchronization
   e.g., The program will deadlock if the release of access is left out

2. Monitors :(Concurrent Pascal, Modula, Mesa)

The idea: encapsulate the shared data and it operations to restrict access

A *monitor* is an abstract data type for shared data show the diagram of monitor buffer operation,

- **Example language***:* Concurrent Pascal
  - Concurrent Pascal is Pascal + classes, processes (tasks), monitors, and the queue data type (for semaphores)

Example language: Concurrent Pascal (continued) processes are types

Instances are statically created by declarations

- An instance is ―started‖ by init, which allocate its local data and begins its execution
- Monitors are also types Form:

type some_name = monitor (formal parameters)

shared variables , local procedures

exported procedures (have entry in definition) initialization code

Competition Synchronization with Monitors:

- Access to the shared data in the monitor is limited by the implementation to a single process at a time; therefore, mutually exclusive access is inherent in the semantic definition of the monitor
- Multiple calls are queued

- Cooperation Synchronization with Monitors:

- Cooperation is still required - done with semaphores, using the queue data type and the built-in operations, delay (similar to send) and continue (similar to release)

- delay takes a queue type parameter; it puts the process that calls it in the specified queue and removes its exclusive access rights to the monitor's data structure

- Differs from send because delay always blocks the caller

- continue takes a queue type parameter; it disconnects the caller from the monitor, thus freeing the monitor for use by another process.

-It also takes a process from the parameter queue (if the queue isn't empty) and starts it, Differs from release because it always has some effect (release does nothing if the queue is empty)

Java Threads

• The concurrent units in Java are methods named run

– A run method code can be in concurrent execution with other such methods

– The process in which the run methods execute is called a *thread*

```
Class myThread extends Thread
public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Controlling Thread Execution

• The Thread class has several methods to control the execution of threads

The yield is a request from the running thread to voluntarily surrender the processor

– The sleep method can be used by the caller of the method to block the thread

– The join method is used to force a method to delay its execution until the run method of another thread has completed its execution

Thread Priorities

• A thread_s default priority is the same as the thread that create it

– If main creates a thread, its default priority is NORM_PRIORITY

• Threads defined two other priority constants, MAX_PRIORITY and MIN_PRIORITY

• The priority of a thread can be changed with the methods setPriority

Cooperation Synchronization with Java Threads

• Cooperation synchronization in Java is achieved via wait, notify, and notifyAll methods

– All methods are defined in Object, which is the root class in Java, so all objects inherit them

• The wait method must be called in a loop

• The notify method is called to tell one waiting thread that the event it was waiting has happened

• The notifyAll method awakens all of the threads on the object_s wait list

Java's Thread Evaluation

• Java_s support for concurrency is relatively simple but effective

• Not as powerful as Ada_s tasks

C# Threads

- Loosely based on Java but there are significant differences

• Basic thread operations

– Any method can run in its own thread

– A thread is created by creating a Thread object

– Creating a thread does not start its concurrent execution; it must be requested through the Start method

– A thread can be made to wait for another thread to finish with Join

– A thread can be suspended with Sleep

– A thread can be terminated with Abort

Synchronizing Threads

- Three ways to synchronize C# threads

– The Interlocked class

• Used when the only operations that need to be synchronized are incrementing or decrementing of an integer

– The lock statement

• Used to mark a critical section of code in a thread lock (expression) {… }

– The Monitor class

• Provides four methods that can be used to provide more

EXCEPTION HANDLING

In a language without exception handling:

When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated

In a language with exception handling:

Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing. Many languages allow programs to trap input/ output errors (including EOF)

Definition 1:

An exception is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing

Definition 2: The special processing that may be required after the detection of an exception is called exception handling

Definition 3: The exception handling code unit is called an exception handler

Definition 4: An exception is raised when its associated event occurs

A language that does not have exception handling capabilities can still define, detect, raise, and handle exceptions

- **Alternatives:**
  1. Send an auxiliary parameter or use the return value to indicate the return status of a Subprogram
     - e.g., C standard library functions
  2. Pass a label parameter to all subprograms (error return is to the passed label)
     - e.g., FORTRAN
  3. Pass an exception handling subprogram to all subprograms

Advantages of Built-in Exception Handling:

1. Error detection code is tedious to write and it clutters the program
2. Exception propagation allows a high level of reuse of exception handling code

Design Issues for Exception Handling:

1. How and where are exception handlers specified and what is their scope?
2. How is an exception occurrence bound to an exception handler?
3. Where does execution continue, if at all, after an exception handler completes its execution?
4. How are user-defined exceptions specified?

5. Should there be default exception handlers for programs that do not provide their own?
6. Can built-in exceptions be explicitly raised?
7. Are hardware-detectable errors treated as exceptions that can be handled?
8. Are there any built-in exceptions?
7. How can exceptions be disabled, if at all?

PL/I Exception Handling

- Exception handler form:

EX:

```
ON condition [SNAP]
    BEGIN;
    ...
    END;
```

  - condition is the name of the associated exception
  - SNAP causes the production of a dynamic trace to the point of the exception
- Binding exceptions to handlers
  - It is dynamic--binding is to the most recently executed ON statement
- Continuation
  - Some built-in exceptions return control to the statement where the exception was raised
  - Others cause program termination
  - User-defined exceptions can be designed to go to any place in the program that is labeled
  - Other design choices:
    - User-defined exceptions are defined with:
      CONDITION exception_name
    - Exceptions can be explicitly raised with:

    - Built-in exceptions were designed into three categories:
      a. Those that are enabled by default but could be disabled by user code
      b. Those that are disabled by default but could be enabled by user code
      c. Those that are always enabled
- Evaluation
  - The design is powerful and flexible, but has the following problems:
    a. Dynamic binding of exceptions to handler makes programs difficult to write and to read
    b. The continuation rules are difficult to implement and they make programs hard
    to read

Ada Exception Handling

Def: The frame of an exception handler in Ada if either a subprogram body, a package body, a task, or a block

- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters
- Handler form:

```
exception
 when exception_name {| exception_name}
     => statement_sequence
 ...
 when ..
 ...
 [when others =>
     statement_sequence]
```

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

- Binding Exceptions to Handlers
  - If the block or unit in which an exception is raised does not have a handler for that exception, the exception is propagated elsewhere to be handled

1. Procedures - propagate it to the caller
2. Blocks - propagate it to the scope in which it occurs
3. Package body - propagate it to the declaration part of the unit that declared the package (if it is a library unit (no static parent), the program is terminated)
4. Task - no propagation; if it has no handler, execute it; in either case, mark it "completed"

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

- **User-defined Exceptions**:

```
exception_name_list : exception;
raise [exception_name]
```

(the exception name is not required if it is in a handler--in this case, it propagates the same exception)

- Exception conditions can be disabled with:

```
pragma SUPPRESS(exception_list)
```

- **Predefined Exceptions**:

CONSTRAINT_ERROR - index constraints, range constraints, etc.

NUMERIC_ERROR - numeric operation cannot return a correct value, etc.

PROGRAM_ERROR - call to a subprogram whose body has not been elaborated

STORAGE_ERROR - system runs out of heap

TASKING_ERROR - an error associated with tasks

- **Evaluation**

  - The Ada design for exception handling embodies the state-of-the-art in language design in 1980

  - A significant advance over PL/I

- Ada was the only widely used language with exception handling until it was added to C++

C++ Exception Handling :

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

C++ Exception Handlers

•Exception Handlers Form:

```
try {
-- code that is expected to raise an exception
}
catch (formal parameter) {
-- handler code
}
...
catch (formal parameter) {
-- handler code
}
```

The catch Function

•catch is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique

•The formal parameter need not have a variable

–It can be simply a type name to distinguish the handler it is in from others

•The formal parameter can be used to transfer information to the handler

•The formal parameter can be an ellipsis, in which case it handles all

Exception Handling in Java

- Based on that of C++, but more in line with OOP philosophy

•All exceptions are objects of classes that are descendants of the Throwable class

Classes of Exceptions

- The Java library includes two subclasses of Throwable:

–Error

- Thrown by the Java interpreter for events such as heap overflow
- Never handled by user programs

–Exception

- User-defined exceptions are usually subclasses of this
- Has two predefined subclasses, IOException and

RuntimeException (e.g., ArrayIndexOutOfBoundsException and NullPointerException

Java Exception Handlers

- Like those of C++, except every catch requires a named parameter and all parameters must be descendants of Throwable

•Syntax of try clause is exactly that of C++

•Exceptions are thrown with throw, as in C++, but often the throw includes the new operator to create the object, as in: throw new MyException ();

Binding Exceptions to Handlers

•Binding an exception to a handler is simpler in Java than it is in C++

–An exception is bound to the first handler with a parameter is the same class as the thrown object or an ancestor of it

•An exception can be handled and rethrown by including a throw in the handler (a handler could also throw a different exception)

Continuation

•If no handler is found in the method, the exception is propagated to the method_s caller

•If no handler is found (all the way to main), the program is terminated

•To ensure that all exceptions are caught, a handler can be included in any try construct that catches all exceptions

–Simply use an Exception class parameter

–Of course, it must be the last in the try construct

Checked and Unchecked Exceptions

- The Java throws clause is quite different from the throw clause of C++

•Exceptions of class Error and RunTimeException and all of their descendants are called unchecked exceptions; all other exceptions are called checked exceptions

•Checked exceptions that may be thrown by a method must be either:

–Listed in the throws clause, or Handled in the method

Other Design Choices

•A method cannot declare more exceptions in its throws clause than the method it overrides

•A method that calls a method that lists a particular checked exception in its throws clause has three alternatives for dealing with that exception:

–Catch and handle the exception

–Catch the exception and throw an exception that is listed in its own throws clause

–Declare it in its throws clause and do not handle it

The finally Clause

•Can appear at the end of a try construct

•Form:

```
finally {
...
}
```

•Purpose: To specify code that is to be executed, regardless of what happens in the try construct

Example

•A try construct with a finally clause can be used outside exception handling

```
try {
for (index = 0; index < 100; index++) {
…
if (…) {
return;
} //** end of if
} //** end of try clause
finally {
…
} //** end of try construct
```

LOGIC PROGRAM PARADIGM:

- Based on logic and declarative programming 60‘s and early 70‘s, Prolog (Programming in logic, 1972) is the most well known representative of the paradigm.
- Prolog is based on Horn clauses and SLD resolution
- Mostly developed in fifth generation computer systems project
- Specially designed for theorem proof and artificial intelligence but allows general purpose computation.
- Some other languages in paradigm: ALF, Frill, G¨odel,, Mercury, Oz, Ciao, _Prolog, datalog, and CLP languages

Constrain Logic Programming:

Clause: disjunction of universally quantified literals, 8(L1 _ L2 _ ... _ Ln)

A logic program clause is a clause with exactly one positive literal

8(A _ ¬A1 _ ¬A2... _ ¬An) _

8(A (A1 ^ A2... ^ An)

A goal clause: no positive literal

8(¬A1 _ ¬A2... _ ¬An)

Proof: by refutation, try to un satisfy the clauses with a goal clause G. Find 9(G).

Linear resolution for definite programs with constraints and selected atom.

CLP on first order terms. (Horn clauses).

Unification. Bidirectional.

Backtracking. Proof search based on trial of all matching clauses

Prolog terms:

Atoms:

- 1 Strings with starting with a small letter and consist of ○ [a-zA-Z 0-9]*
  - a aDAM a1 2
- 2 Strings consisting of only punctuation
- *** .+. .<>.
- 3 Any string enclosed in single quotes (like an arbitrary string) ○ ‘ADAM‘ ‘Onur Sehitoglu‘‘2 * 4 < 6‘
- Numbers
  - 1234 12.32 12.23e-10

Variables:

- Strings with starting with a capital letter or and consist of
- [a-zA-Z 0-9]*
- Adam adam A093
- is the universal match symbol. Not variable

Structures:

Starts with an atom head have one or more arguments (any term) enclosed in parenthesis, separated by comma structure head cannot be a variable or anything other than atom.

a(b) a(b,c) a(b,c,d) ++(12) +(*) *(1,a(b)) ‘hello world‘(1,2) p

X(b) 4(b,c) a() ++() (3) × some structures defined as infix:

+(1,2) _ 1+2 , :-(a,b,c,d) _ a :- b,c,d

Is(X,+(Y,1)) _ X is X + 1

Static sugars:

Prolog interpreter automatically maps some easy to read syntax into its actual structure.

List: [a,b,c] _ .(a,.(b,.(c,[])))

Head and Tail: [H|T] _ .(H,T)

String: "ABC" _ [65,66,67] (ascii integer values) use display (Term). to see actual structure of the term

Unification:

Bi-directional (both actual and formal argument can be instantiated).

1. if S and T are atoms or number, unification successful only if

S = T

2 . if S is a variable, S is instantiated as T, if it is compatible with current constraint store (S is instantiated to another term, they are unified)

3 if S and T are structures, successful if:

head of S = head of T

they have same arity

unification of all corressponding terms are successful.

S: list of structures, P current constraint store

s 2 S, arity(s): number of arguments of structure,

s 2 S, head(s): head atom of the structure,

s 2 S, argi (s): i th argument term of the structure,

p _ P: p is consistent with current constraint store.

S _ T;P =

(S,T 2 A _ S,T 2 N) ^ S = T ! true's S 2 V ^ S _ T |= P ! true; S _ T ^ P T 2 V ^ S _ T |= P ! true; S _ T ^ P S,T 2 S ^ head(S) = head(T) ^ arity(S) = arity(T) !

8i , argi (S) _ argi (T); P

Unification Example:

X = a ! pwith X = a

a(X,3) = a(X,3,2) ! ×

a(X,3) = b(X,3) ! ×

a(X,3) = a(3,X) ! pwith X = 3

a(X,3) = a(4,X) ! × a(X,b(c,d(e,f))) = a(b(c,Y),X) ! X = b(c,d(e,f)), Y = d(e,F)

Declarations:

Two types of clauses:

p1(arg1, arg2, ...) :- p2(args,...) , p3(args,...) .means if p2 and p3 true, then p1 is true. There can be arbitrary number of (conjunction of) predicates at right hand side.

p(arg1, arg2, ...) .sometimes called a fact. It is equivalent to:

p(arg1, arg2, ...) :- true.

p(args) :- q(args) ; s(args) .

Is disjunction of predicates. q or s implies p. Equivalent to:

p(args) :- q(args).

p(args) :- s(args).

A prolog program is just a group of such clauses.

Lists Example:

% list membership

memb(X, [X| Re s t]) .

memb(X, [| Re s t]) :- memb(X, Re s t).

% concatenation

conc ([] ,L,L).

conc ([X|R] , L , [X|R and L]) :- conc (R, L, R and L).

% second list starts with first list prefix of ([],).

Prefix of ([X|Rx], [X|Ry]) :- prefix of (Rx,Ry).

% second list contains first list

Sublist (L1,L2) :- prefix of (L1,L2).

Sublist (L,[|R]):- sublist(L,R).

Procedural Interpretation:

For goal clause all matching head clauses (LHS of clauses) are kept as backtracking points (like a junction in maze search) Starts from first match. To prove head predicate, RHS predicates need to be proved recursively. If all RHS predicates are proven, head predicate is proven. When fails, prolog goes back to last backtracking point and tries next choice. When no backtracking point is left, goal clause fails. All predicate matches go through unification so goal clause variables can be instantiated.

Arthematic and Operations:

X = 3+1 is not an arithmetic expression!

operators (is) force arithmetic expressions to be evaluated all variables of the operations needs to be instantiated.

12 is 3+X does not work!

Comparison operators force LHS and RHS to be evaluated:

X>Y, X<Y, X>=Y, X =< Y, X =:= Y, X == Y

is operator forces RHS to be evaluated: X is Y+3*Y Y needs to have a numerical value when search hits this expression. Note that X is X+1 is never successful in Prolog. Variables are instantiated once.

Greatest Common Divisor:

$gcd(m, n) = gcd(n, m - n)$ if $n < m$

$gcd(m, n) = gcd(n, m)$ if $m < n$

gcd (X,X,X) .

gcd (X,Y,D) :- X < Y, Y1 is Y-X, gcd (X,Y1,D).

gcd (X,Y,D) :- Y < X, gcd (Y, X, D).

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

UNIT-V

FUNCTIONAL PROGRAMMING LANGUAGES

Functional Programming Languages:

- 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:

Def: 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 f(x) x * x * x for the function cube (x) = x * x * x   functions.

- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression e. g. (f(x) x * x * x)(3) which evaluates to 27.

Functional Forms:

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

1.Function Composition:

A functional form that takes two functions as parameters and yields a function whose result is a function whose value is the first actual parameter function applied to the result of the application of the second Form: h(f)° g which means h (x) f (g (x))

2. Construction:

A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter

Form: [f, g]

For f (x) = x * x * x and g (x) = x + 3,

[f, g] (4) yields (64, 7)

3. Apply-to-all:

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

Form:

For h (x) =x * x * x

f(h, (3, 2, 4)) yields (27, 8, 64)

LISP –

LISP is the first functional programming language, it contains two forms those are **1**. **Data object types**: originally only atoms and lists

2. List form: parenthesized collections of sub lists and/or atoms

e.g., (A B (C D) E)

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
- In an FPL, the evaluation of a function always produces the same result given the same parameters
- This is called referential transparency

A Bit of LISP:

- Originally, LISP was a type less language. There were only two data types, atom and list
- LISP lists are stored internally as single-linked lists
- Lambda notation is used to specify functions and function definitions, function applications, and data all 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

Scheme:

- A mid-1970s dialect of LISP, designed to be cleaner, more modern, and simpler version than the contemporary dialects of LISP, Uses only static scoping

- Functions are first-class entities, They can be the values of expressions and elements of lists, They can be assigned to variables and passed as parameters

- **Primitive Functions:**

1. Arithmetic: +, -, *, /, ABS, SQRT

Ex: (+ 5 2) yields 7

2. QUOTE: -takes one parameter; returns the parameter without evaluation

- **QUOTE** is required because the Scheme interpreter, named **EVAL,** always evaluates parameters to function applications before applying the function. **QUOTE** is used to avoid parameter evaluation when it is not appropriate

- **QUOTE** can be abbreviated with the apostrophe prefix operator

e.g., '**(A B)** is equivalent to **(QUOTE (A B))**

3. CAR takes a list parameter; returns the first element of that list

e.g., (CAR '(A B C)) yields **A**

(CAR '((A B) C D)) yields **(A B)**

4. CDR takes a list parameter; returns the list after removing its first element e.g., (CDR '(A B C)) yields (B C)

(CDR '((A B) C D)) yields (C D)

5. CONS takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result

e.g., (CONS 'A '(B C)) returns (A B C)

6. LIST - takes any number of parameters; returns a list with the parameters as elements

- Predicate Functions: (#T and () are true and false)

1. EQ? takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same

e.g.,(EQ? 'A 'A) yields #T

(EQ? 'A '(A B)) yields ()

Note that if EQ? is called with list parameters, the result is not reliable Also, EQ? does not work for numeric atoms

2. LIST? takes one parameter; it returns #T if the parameter is an list; otherwise ()

3. NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise ()

Note that NULL? returns #T if the parameter is ()

4. Numeric Predicate Functions

=, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?

5. Output Utility Functions:

(DISPLAY expression)

(NEWLINE)

- Lambda Expressions
  - Form is based on notation

    e.g.,

    (LAMBDA (L) (CAR (CAR L))) L is called a bound variable

  - Lambda expressions can be applied

    e.g.,

    ((LAMBDA (L) (CAR (CAR L))) '((A B) C D))

- A Function for Constructing Functions

  DEFINE - Two forms:

  1. To bind a symbol to an expression

     EX:

     (DEFINE pi 3.141593)

     (DEFINE two_pi (* 2 pi))

  2. To bind names to lambda expressions

     EX:

     (DEFINE (cube x) (* x x x))

  - Example use:

    (cube 4)

  - **Evaluation process (for normal functions)**:
    1. Parameters are evaluated, in no particular order
    2. The values of the parameters are substituted into the function body
    3. The function body is evaluated
    4. The value of the last expression in the body is the value of the function

       (Special forms use a different evaluation process)

Control Flow:

- 1. Selection- the special form, IF

  (IF predicate then_exp else_exp)

  e.g.,

  (IF (<> count 0)

```
(/ sum count)
0 )
```

ML :

- A static-scoped functional language with syntax, that is closer to Pascal than to LISP

- Uses type declarations, but also does type inferencing to determine the types of undeclared variables (See Chapter 4)

- It is strongly typed (whereas Scheme is essentially type less) and has no type coercions

- Includes exception handling and a module facility for implementing abstract data types

- Includes lists and list operations

- The val statement binds a name to a value (similar to DEFINE in Scheme)

- Function declaration form: fun function_name (formal_parameters) = function_body_expression;

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

- Functions that use arithmetic or relational operators cannot be polymorphic--those with only list operations can be polymorphic

Haskell:

- Similar to ML (syntax, static scoped, strongly typed, type inferencing)

- Different from ML (and most other functional languages) in that it is PURELY functional

  (e.g., no variables, no assignment statements, and no side effects of any kind)

- *Most Important Features*

- Uses lazy evaluation (evaluate no subexpression until the value is needed,

Has ―list comprehensions,‖ which allow it to deal with infinite lists

Examples

1. ***Fibonacci numbers*** (illustrates function definitions with different parameter forms)

```
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

2. Factorial (illustrates guards)

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

a guard

3. List operations

- List notation: Put elements in brackets

e.g., directions = [north, south, east, west]

e.g., #directions is 4

- Arithmetic series with the ..operator

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

- Catenation is with +

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

[1, 3, 5, 7]

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

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

- ***Examples:***

```
product [ ] = 1
product (a:x) = a * product x
fact n = product [1..n]
```

4. List comprehensions: set notation e.g.,

[n * n | n ¬ [1..20]]

defines a list of the squares of the first 20

positive integers

factors n = [i | i [1..n div 2],

n mod i == 0]

This function computes all of the factors of its , given parameter

Quicksort:

sort [ ] = [ ]

sort (a:x) = sort [b | b ¬ x; b <= a]

++ [a] ++

sort [b | b ¬ x; b > a]

***5.* Lazy evaluation**

- Infinite lists

e.g.,

positives = [0..]

squares = [n * n | n ¬ [0..]]

(only compute those that are necessary)

e.g.,

member squares 16 would return True ,The member function could be written as: member [] b = False member (a:x) b = (a == b) || member x

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

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

Applications of Functional Languages:

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

- programming at a significant number of universities

Comparing Functional and Imperative Languages

Imperative Languages:

- Efficient execution
- Complex semantics
- Complex syntax
- Concurrency is programmer designed

- Functional Languages:

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

Scripting languages

Pragmatics

- *Scripting* is a paradigm characterized by:

  -use of scripts to glue subsystems together;

  -rapid development and evolution of scripts;

  -modest efficiency requirements;

-very high-level functionality in application-specific areas.

- A software system often consists of a number of subsystems controlled or connected by a script.

In such a system, the script is said to glue the sub systems together

COMMON CHARACTERISTICS OF SCRIPTING LANGUAGES

- Both batch and interactive use
- Economy of expressions
- Lack of declaration; simple scoping rules
- Flexible dynamic typing
- Easy access to other programs
- High level data types
- Glue other programs together
- Extensive text processing capabilities
- Portable across windows, unix ,mac

PYTHON

- PYTHON was designed in the early 1990s by Guido van Rossum.
- PYTHON borrows ideas from languages as diverse as PERL, HASKELL, and the object-oriented languages, skillfully integrating these ideas into a coherent whole.

PYTHON scripts are concise but readable**,** and highly expressive

Python is extensible: if we invoke how to program in C, it is easy to add new built in function or module to the interpreter, either to perform critical operations at maximum speed of to link python programs to libraries that may only be available in binary form

Python has following characteristics.

- Easy to learn and program and is object oriented.
- Rapid application development
- Readability is better
- It can work with other languages such as C,C++ and Fortran
- Powerful interpreter

- Extensive modules support is available

Values and types

- PYTHON has a limited repertoire of primitive types: integer, real, and complex Numbers.
- It has no specific character type; single-character strings are used instead.
- Its boolean values (named False and True) are just small integers.
- PYTHON has a rich repertoire of composite types: tuples, strings, lists, dictionaries, and objects.

Variables, storage, and control

- PYTHON supports global and local variables.
- Variables are not explicitly declared, simply initialized by assignment.
- PYTHON adopts reference semantics. This is especially significant for mutable values, which can be selectively updated.

Primitive values and strings are immutable; lists, dictionaries, and objects are mutable; tuples are mutable if any of their components are mutable

- PYTHON's repertoire of commands include assignments, procedure calls, con-ditional (if-butnotcase-) commands, iterative (while- and for-) commands, and exception-handling commands.

Pythons reserved words are:

and assert break class continue def del
elif
else except exec finally for from global if
import in is lambda not or pass
print
raise return try while yield

Dynamically typed language:

Python is a dynamically typed language. Based on the value, type of the variable is during the execution of the program.

Python (dynamic)

```
C = 1
C = [1,2,3]
C(static)
Double c; c = 5.2;
C = —a string….‖
```

Strongly typed python language:

Weakly vs. strongly typed python language differs in their automatic conversions.

Perl (weak)

```
$b = `1.2`
$c = 5 * $b;
```

Python (strong)

```
b =`1.2`
c= 5* b;
```

PYTHON if- and while-commands are conventional

Bindings and scope

- A PYTHON program consists of a number of modules, which may be grouped into packages.
- Within a module we may initialize variables, define procedures, and declare classes
- Within a procedure we may initialize local variables and define local procedures.
- Within a class we may initialize variable components and define procedures (methods).
- PYTHON was originally a dynamically-scoped language, but it is now statically scoped

In python, variables defined inside the function are local to that function. In order to change them as global variables, they must be declared as global inside the function as given below.

```
S = 1

Def  myfunc(x,y);
```

```
Z = 0
Global  s;
 S = 2
Return   y-1 , z+1;
```

Procedural abstraction

- PYTHON supports function procedures and proper procedures.
- The only difference is that a function procedure returns a value, while a proper procedure returns nothing.

Since PYTHON is dynamically typed, a procedure definition states the name but not the type of each formal parameter

Python procedure

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

Python procedure with Dynamic Typing

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

Data Abstraction

- PYTHON has three different constructs relevant to data abstraction: packages ,modules , and classes

- Modules and classes support encapsulation, using a naming convention to distinguish between public and private components.
- A Package is simply a group of modules
- A Module is a group of components that may be variables, procedures, and classes
- A Class is a group of components that may be class variables, class methods, and instance methods.
- A procedure defined in a class declaration acts as an instance method if its first formal parameter is named self and refers to an object of the class being declared. Otherwise the procedure acts as a class method.

Separate Compilation

- PYTHON modules are compiled separately.
- Each module must explicitly import every other module on which it depends
- Each module's source code is stored in a text file. E g: program.py
- When that module is first imported, it is compiled and its object code is stored in a file named program.pyc
- Compilation is completely automatic
- The PYTHON compiler does not reject code that refers to undeclared identifiers. Such code simply fails if and when it is executed
- The compiler will not reject code that might fail with a type error, nor even code that will certainly fail, such as:

```
Def fail (x):

Print x+1, x[0]
```

Module Library

- PYTHON is equipped with a very rich module library, which supports string handling, markup, mathematics, and cryptography, multimedia, GUIs, operating system services, internet services, compilation, and so on.
- Unlike older scripting languages, PYTHON does not have built-in high-level string processing or GUI support, so module library provides it.