LECTURE NOTES

PRINCIPLES OF PROGRAMMING LANGUAGES

UNIT - I

Reasons for Studying Concepts of Programming Languages

Increased ability to express ideas.

- Itisbelievedthatthedepthatwhichwethinkisinfluencedbytheexpressivepowerofthe language in which we communicate our thoughts. It is difficult for people to conceptualize structures they can‘t describe, verbally or inwriting.
- 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 languagechoices.

understand significance of implementation

- Understandingofimplementationissuesleadstoanunderstandingofwhylanguagesare designed the way theyare.
- Thisinturnleadstotheabilitytousealanguagemoreintelligently,asit wasdesignedto be used.

Ability to design new languages

- The more languages you gain knowledge of, the better understandingof programming languages concepts youunderstand.

Overall advancement of computing

- Insomecases,alanguagebecamewidelyused,atleast inpart,b/cthosein positions to choose languages were not sufficiently familiar with P/L concepts.
- ManybelievethatALGOL60wasabetterlanguagethanFortran;however, Fortran was most widely used. It is attributed to the fact that the programmers and managers didn‘t understand the conceptual design of ALGOL60.
- Do you think IBM has something to do withit?

Programming Domains

Scientific applications

- In the early 40s computers were invented for scientificapplications.
- The applications require large number of floating pointcomputations.
- Fortran was the first language developed scientificapplications.
- ALGOL 60 was intended for the same use.

Business applications

- The first successful language for business wasCOBOL.
- Produce reports, use decimal arithmetic numbers andcharacters.
- The arrival of PCs started new ways for businesses to usecomputers.
- Spreadsheets and database systems were developed forbusiness.

Artificial intelligence

– Symbolic rather than numeric computations aremanipulated.

– Symbolic computation is more suitably done with linked lists thanarrays.

– LISP was the first widely used AI programminglanguage.

Systems programming

– The O/S and all of the programming supports tools are collectively known asits systemsoftware.

– Need efficiency because of continuoususe.

Scripting languages

– Put a list of commands, called a script, in a file to beexecuted.

– 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 requestingbrowser.

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 computer than the users.

Because ease of maintenance is determined in large part by the readability ofprograms, 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 tendto learn a subset of the language and ignore its other features. **—**ALGOL60

– Multiplicity of features is also a complicating characteristic **—**having morethan one way to accomplish a particularoperation.

– 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.

– Operatoroverloadingwhereasingleoperatorsymbolhasmorethanonemeaning.

– 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 itsensibly.

Orthogonality:

- Makes the language easy to learn and read.
- Meaningiscontextindependent.Pointersshouldbeabletopointtoanytypeofvariableor data structure. The lack of orthogonality leads to exceptions to the rules of the language.
- Arelativelysmallsetofprimitiveconstructscanbecombinedinarelativelysmallnumber of ways to build the control and data structures of thelanguage.
- Every possible combination is legal and meaningful.
- 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

– Itbecamewidelyrecognizedthatindiscriminateuseofgotostatementsseverelyreduced programreadability.

– 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 restrictionsontheuseofgotos,sowritinghighlyreadableprogramsinthoselanguages wasdifficult.

– Since then, languages have included sufficient controlstructures.

– The control statement design of a language is now a less important factor in readability than it was in thepast.

Data Types and Structures

– The presence of adequate facilities for defining data types and data structures ina language is another significant aid toreliability.

– Ex: Boolean type.

timeout=1 or timeout =true

Syntax Considerations

- The syntax of the elements of a language has a significant effect onreadability.
- The following are examples of syntactic design choices that affectreadability:

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 isdifficulttodeterminewhichgroupisbeingended.Fortran95allowsprogrammers to use special names as legal variablenames.

Form and Meaning: Designing statements so that their appearance at least partially indicates their purpose is an obvious aid toreadability.

- 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

– Asmallernumberofprimitiveconstructsandaconsistentsetofrulesforcombining them is much better than simply having a large number ofprimitives.

Support for abstraction

– **Abstraction** means the ability to define and then use complicated structuresor operations in ways that allow many of the details to beignored.

– Aprocessabstractionistheuseofasubprogramtoimplementasortalgorithmthatis requiredseveraltimesinaprograminsteadofreplicatingitinallplaceswhereitis needed.

Expressivity

– Itmeansthatalanguagehasrelativelyconvenient,ratherthancumbersome,waysof specifyingcomputations.

– Ex:++count⇔count=count+1//moreconvenientandshorter

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 correctivemeasures, and then continue is a great aid toreliability.

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 alanguage.

Readability and writability: Both readability and writability influence reliability.

Cost

– Categories

– Training programmers to use language

– **Writingprograms—**Writability

– Compilingprograms

– Executingprograms

– Languageimplementationsystem—Freecompilersisthekey,successofJava

– **Reliability**, does the softwarefail?

– **Maintaining**programs:Maintenancecostscanbeashighastwotofourtimesas much as development costs.

– Portability—standardizationofthelanguage‖,Generality(theapplicabilitytoa wide range ofapplications)

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 samememory

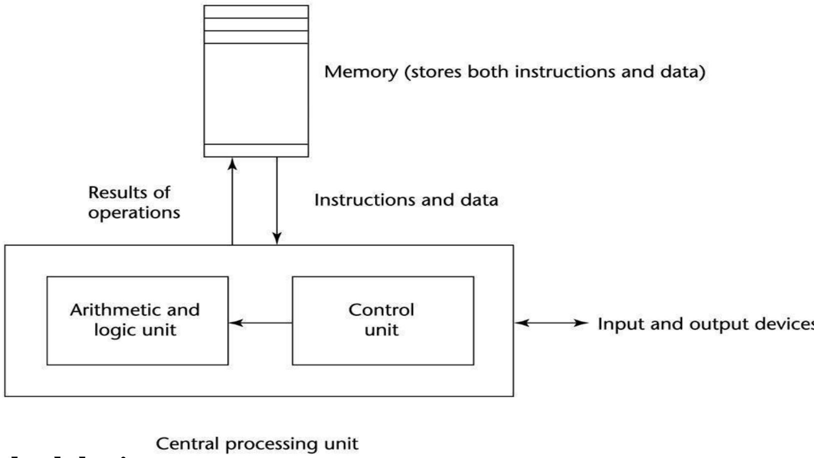
– Memory is separate fromCPU

– Instructions and data are piped from memory toCPU

– Results of operations in the CPU must be moved back tomemory

– Basis for imperativelanguages

- Variables model memorycells
- Assignment statements modelpiping
- Iteration isefficient



Programming methodologies

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

Late 1960s: People efficiency became important; readability, better control structures

– Structuredprogramming

– Top-down design and step-wiserefinement

Late 1970s: Process-oriented to data-oriented

– dataabstraction

Middle 1980s: Object-oriented programming

Language Categories

Imperative

– Central features are variables, assignment statements, anditeration

– 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 specialorder

– Prolog

Object-oriented

– Encapsulate data objects withprocessing

– Inheritance and dynamic typebinding

– Grew out of imperativelanguages

– C++,Java

<u>Programming Environments</u>

The collection of tools used in software development

UNIX

– An older operating system and toolcollection

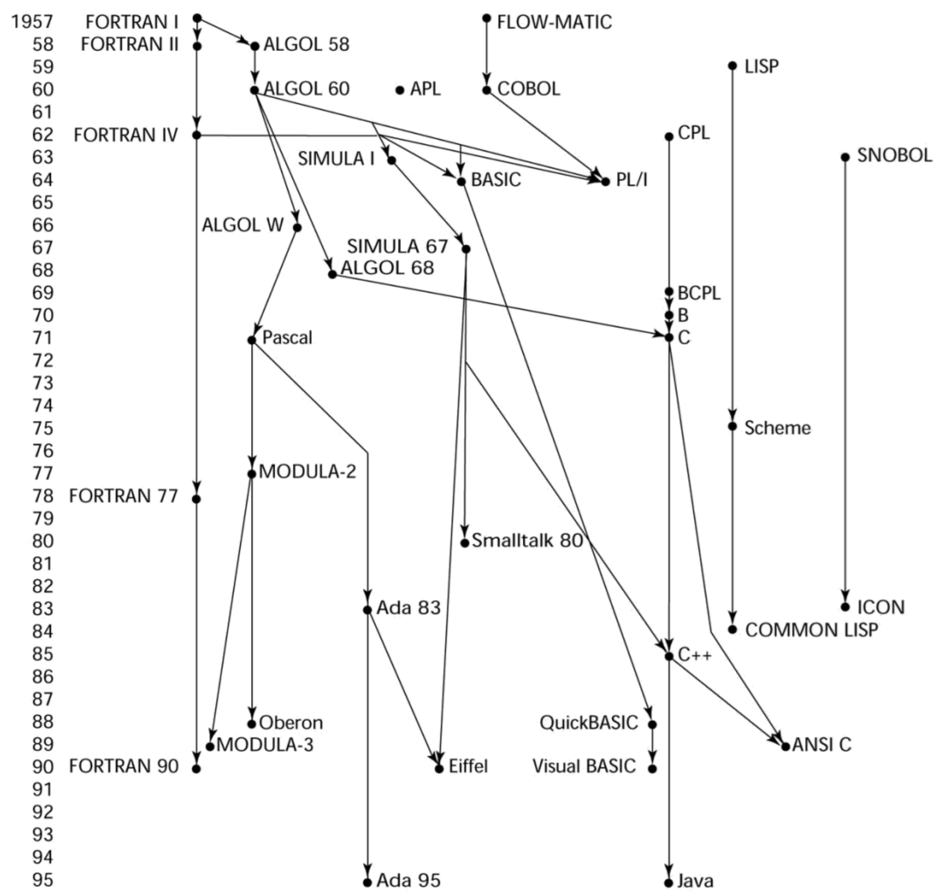
Borland JBuilder

– An integrated development environment forJava

Microsoft Visual Studio.NET

– A large, complex visualenvironment

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



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**?

- Other languagedesigners
- Implementers
- 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**isthelowestlevelsyntacticunitofalanguage(e.g.,*,sum,begin)A**token**isacategory of lexemes (e.g., identifier)

Formal approaches to describing syntax:

Recognizers - used in compilers

Generators - what we'll study

Language recognizers:

Suppose we have a language L that uses an alphabet ∑ of characters. To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet ∑. R would indicate whether a given input string was or was not in L. In effect, R would either accept or reject the given string. Such devices are like filters, separating legal sentences from those that are incorrectly formed. If R,whenfedanystringofcharactersover∑,acceptsitonlyifitisinL,thenRisadescriptionofL.

Becausemostusefullanguagesare,forallpracticalpurposes,infinite,thismightseemlike a lengthy and ineffective process. Recognition devices, however, are not used to enumerate all of the sentences of a language—they have a differentpurpose.

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 otherhand, it is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of thegenerator.

Context-Free Grammars

- Developed by Noam Chomsky in themid-1950s
- Language generators, meant to describe the syntax of naturallanguages
- Define a class of languages called *context-free* Languages.
- Arulehasaleft-handside(LHS)andaright-handside(RHS),andconsistsof*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> | constAn 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>

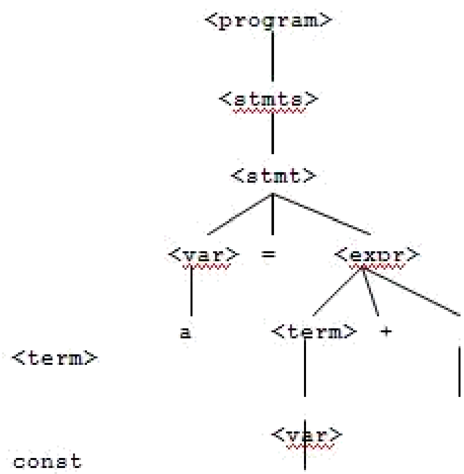
<stmt> -><var> = <expr>
<var> -> a | b | c | d

<expr> -><term> + <term> | <term> - <term><term>
-><var> | constAn example derivation:
<program> =><stmts> =><stmt>

=><var> = <expr> => a = <expr>
=> a = <term> + <term>

=> a = <var> +<term>
=> a = b + <term> =>a
= b + const

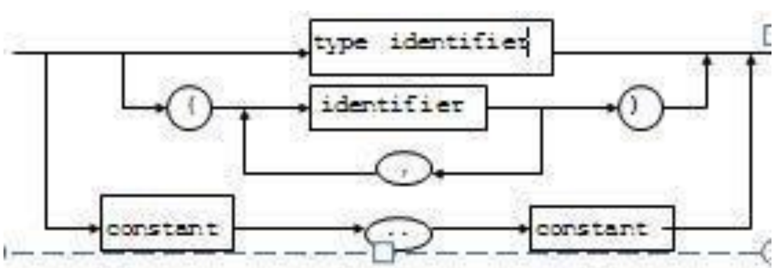
Every string of symbols in the derivation is a *sentential form* A *sentence* is a sentential form that has only terminal symbols A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded A derivation may be neither leftmost nor rightmost A *parse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

- **Context-free** but cumbersome (e.g. typechecking)
- **Noncontext-free** (e.g. variables must be declared before they areused

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

- Additions to cfgsto carry some semantic info along through parse tree Primaryvalue ofAGs:

  - Static semanticsspecification

- Compiler design (static semanticschecking)

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

For each grammar symbol x there is a set A(x) of attribute values

Each rule has a set of functions that define certain attributes of the nonterminals in the rule

Each rule has a (possibly empty) set of predicates to check for attribute consistency

LetX0->X1...XnbearuleFunctionsoftheformS(X0)=f(A(X1),...A(Xn))define*synthesized attributes* Functions of the form I(Xj) = f(A(X0), ... , A(Xn)), for i<= j <= n, define *inherited attributes* Initially, there are *intrinsic attributes* on theleaves

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

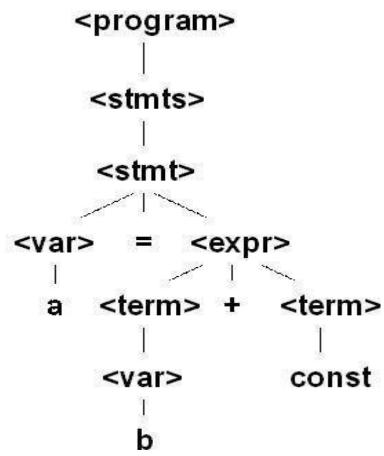
types of the two id's must be the same type of the expression must match it's expected type

BNF:

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

PARSE TREE:

Ahierarchicalrepresentationofaderivation.Everyinternalnodeofaparsetreeislabeled withanonterminalsymbol;everyleafislabeledwithaterminalsymbol.Everysubtreeofaparse tree describes one instance of an abstraction in thesentence



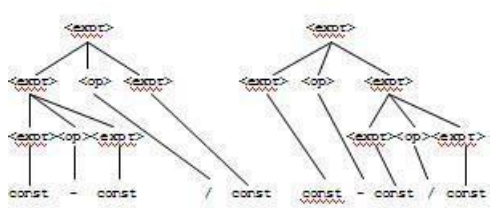
Ambiguous grammar

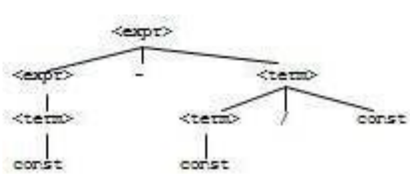
Agrammaris**ambiguous**ifit generatesasententialformthathastwoormoredistinct parse trees Anambiguous expressiongrammar:

<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 expressiongrammar:

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





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

=>const - <term>

=>const - <term> / const

=>const - const / const

Extended BNF (just abbreviations):

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

Put alternative parts of RHSs in parentheses and separate them with vertical bars term> -><term> (+ | -) const

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>

How are attribute values computed?

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

 If all attributes were synthesized, the tree could be decorated in bottom-up order.

 In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used

. <expr>.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_type4.

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

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

Dynamic Semantics

- No single widely acceptable notation or formalism for describingsemantics

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 machinein needed
- A *hardware* pure interpreter would be tooexpensive
- A *software* pure interpreter also hasproblems:

The detailed characteristics of the particular computer would make actions difficult to understand

Such a semantic definition would be machine dependent

A better alternative: A complete computer simulation

Theprocess:

- Build a translator (translates source code to the machine code of an idealizedcomputer)
- Build a simulator for the idealizedcomputer

Evaluation of operational semantics:

- Good if usedinformally
- 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***

Anassertionbeforeastatement(a*precondition*)statestherelationshipsandconstraints among variables that are true at that point inexecution

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

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

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:

P => I (the loop invariant must be true initially)

{I} B {I} (evaluation of the Boolean must not change the validity of I)

{I and B} S {I} (I is not changed by executing the body of the loop)

(I and (not B)) => Q (if I is true and B is false, Q is implied)

The loop terminates (this can be difficult to prove)

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

Imust beweakenoughto besatisfiedpriortothebeginningoftheloop,but whencombined with the loop exit condition, it must be strong enough to force the truth ofthe postcondition

Evaluation of axiomatic semantics*:*

- Developing axioms or inference rules for all of the statements in a language isdifficult

- It is a good tool for correctness proofs, and excellent framework for reasoningabout programs, but it is not as useful for language users and compilerwriters

De-notational Semantics

- Based on recursive function theory The most abstract semantics descriptionmethod Originally developed by Scott andStrachey
- The process of building a de-notational spec for alanguage:

_ Define a mathematical object for each language entity

_ Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

The meaning of language constructs are defined by only the values of the program's variables

The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions

- The *state* of a program is the values of all its current variables s = {<i1, v1>,<i2, v2>,…, <in,vn>}

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

VARMAP (ij, s) = vj

1. DecimalNumbers

<dec_num> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| <dec_num> (0 | 1 | 2 | 3 | 4 |
5 | 6 | 7 | 8 | 9)

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

Mdec (<dec_num> '0') = 10 * Mdec (<dec_num>) Mdec (<dec_num> '1') = 10 * Mdec (<dec_num>) + 1

…

Mdec (<dec_num> '9') = 10 * Mdec (<dec_num>) + 9

2. ExpressionsMe(<expr>, s) = case <expr>of

<dec_num> =>Mdec(<dec_num>, s)

<var> =>

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

else VARMAP(<var>, s)

<binary_expr> =>

if (Me(<binary_expr>.<left_expr>, s) = undef OR Me(<binary_expr>.<right_expr>, s) = undef) then error

else

if (<binary_expr>.<operator> = ë+í then Me(<binary_expr>.<left_expr>, s) + Me(<binary_expr>.<right_expr>, s)

else Me(<binary_expr>.<left_expr>, s) * Me(<binary_expr>.<right_expr>, s)

3 Assignment Statements Ma(x := E, s) = if Me(E, s) = error then error

else s‘ = {<i1‘,v1‘>,<i2‘,v2‘>,...,<in‘,vn‘>}, where for j = 1, 2, ..., n,
vj‘ = VARMAP(ij, s) if ij<> x

= Me (E, s) if ij = x 4 Logical Pretest Loops

Ml(while B do L, s) = if Mb(B, s) = undef then error

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

else if Msl (L, s) = error

then error

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

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

- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions
- Recursion, when compared to iteration, is easier to describe with mathematicalrigor

Evaluation of denotational semantics:

- Can be used to prove the correctness ofprograms
- Provides a rigorous way to think aboutprograms
- Can be an aid to languagedesign
- Has been used in compiler generationsystems

UNIT - 2

Data type Definition:

- Collection of dataobjects
- a set of predefinedoperations

- **descriptor** : collection of attributes for avariable

- **object** :instance of a user-defined (abstract data)type

Data Types:

- **Primitive**
  - not defined in terms of other data types

  - defined in thelanguage
  - often reflect thehardware

- **Structured**
  - built out of othertypes

Integer Types:

- ➢ Usually based onhardware

- ➢ May have severalranges

  - **Java's signed integer sizes**: byte, short, int,long
  - **C/C++** have unsigned versions of the sametypes

  - **Scripting languages** often just have one integertype
  - **Python** has an integer type and a long integer which can get as big as it needsto.

- **RepresentingIntegers:**
  - Can convert positive integers tobase

  - How do you handle negative numbers with only 0s and1s?

– Signbit
– Onescomplement

– Twos complement - this is the one that isused
– Representing negativeintegers.

Representing negative integers:

– Signbit

– Onescomplement

Twos Complement:

- To get the binary representation, take the complement and add1

Floating Point Types:

Model real numbers only an approximation due to round-off error

– Forscientificusesupportatleasttwofloating-pointtypes(e.g.,floatanddouble; sometimesmore)

– The float type is the standard size, usually being stored in four bytes ofmemory.

– Thedoubletypeisprovidedforsituationswherelargerfractionalpartsand/oralarger range of exponents is needed

– Floating-pointvaluesarerepresentedasfractionsandexponents,aformthatisborrowed from scientificnotation

– Thecollectionofvaluesthat canberepresentedbyafloating-pointtype isdefined in terms of precision andrange

– **Precision** is the accuracy of the fractional part of a value, measured as the number ofbits

– **Range**isacombinationoftherangeoffractionsand,moreimportant,therangeof exponents.

– Usually based onhardware

– IEEE Floating-Point Standard754

–32 and 64 bit standards

Representing Real Numbers:

- We can convert the decimal number to base 2 just as we did forintegers
- How do we represent the decimalpoint?
  - fixednumberofbitsforthewholeandfractionalpartsseverelylimitstherangeof values we canrepresent
- Use a representation similar to scientificnotation

IEEE Floating Point Representation:

- Normalize thenumber
  - one bit before decimalpoint
- Use one bit to represent the sign (1 fornegative)
- Use a fixed number of bits for the exponent which is offset to allow for negative exponents

Floating Point Types:

- C, C++ and Java have two floating pointtypes
  - float
  - double
- Most scripting languages have one floating pointtype
  - Python's floating point type is equivalent to a Cdouble
- Some scripting languages only have one kind of number which is a floating pointtype

Fixed Point Types (Decimal) :

- For business applications (money) round-off errors are notacceptable
  - Essential toCOBOL

– .NET languages have a decimal datatype

– store a fixed number of decimaldigits

– Operations generally have to be defined insoftware

– ***Advantage***:accuracy

– ***Disadvantages***: limited range, wastesmemory

C# decimal Type:

– 128-bitrepresentation

– **Range:**1.0×10^{-28} to7.9×10^{28}

– **Precision**: representation is exact to 28 or 29 decimal places (depending on size of number)

– no roundofferror

Other Primitive Data Types:

– **Boolean**

– Rangeofvalues:two elements, onefor ―true‖andonefor ―false‖

– Could be implemented as bits, but often asbytes

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

– ABooleanvaluecouldberepresentedbyasinglebit,butbecauseasingle bitofmemorycannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically abyte.

.

- **Character**
  - Stored as numericcodings
  - Most commonly used coding:ASCII
  - An alternative, 16-bit coding:Unicode
- Complex (Fortran, Scheme,Python)
- ational(Scheme)

Character Strings :

- Values are sequences ofcharacters
- Character string constants are used to label output, and the input and output of allkinds of data are often done in terms ofstrings.
- **Operations:**
  - Assignment andcopying
  - Comparison (=, >,etc.)
  - Catenation
  - Substringreference
  - Patternmatching
  - 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 stringsare complicated by the possibility of string operands of differentlengths.

- **Designissues:**
  - Is it a primitive type or just a special kind ofarray?
  - Should the length of strings be static ordynamic?

Character String Implementations:

- **C andC++**
  - Notprimitive
  - Use char arrays and a library of functions that provideoperations
- **SNOBOL4 (**a string manipulationlanguage)
  - Primitive
  - Many operations, including elaborate patternmatching
- **Java**
  - Stringclass

String Length Options

- **Static**: COBOL, Java's Stringclass
- **Limited Dynamic Length**: C andC++

– a special character is used to indicate the end of a string'scharacters

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

StringImplementation

- **Static length:** compile-timedescriptor
- **Limited dynamic length**: may need run-timedescriptor

- not in C andC++
- **Dynamic length:** needs run-timedescriptor;
  - allocation/deal location is main implementationissue

User-Defined Ordinal Types:

- **Cordinaltype** : range of possible values corresponds to set of positiveintegers
- **Primitive ordinaltypes**
  - integer
  - char
  - boolean
- **User-defined ordinaltypes**
  - enumerationtypes
  - ubrangetypes

Enumeration Types:

- All possible values, which are named constants, are provided in thedefinition

C example:

Enum days {Mon, Tue, wed, Thu, Fri, sat, sun};

- Designissues
  - duplication ofnames
  - coercionrules

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 aname

typedefenum weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} weekday;

Weekday today = Tuesday;

- By default, values are consecutive starting from0.
  - You can explicitly assign values **Enum months {January=1,February,};**

Enumerations in Java 1.5:

- An enumis a new class which extends **java.lang.Enum**and implementsComparable
  - Get type safety and compile-timechecking
  - Implicitly public, static andfinal
  - Can use either == or equals tocompare
  - toString and valueOf are overridden to make input and outputeasier

Java enum Example:

- **Defining an enumtype**

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

- **Declaring an enumvariable**

  Season season = Season.WINTER;

- to String gives you the **string representation of thename**

  System.out.println (season);

➢ printsWINTER

– **valueOf**letsyouconvertaStringtoanenumSeason=valueOf(―SPRING‖);

Sub range Types:

– A contiguous subsequence of an ordinaltype

Example:

– **Ada'sdesign:**

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

Subrangetypesenhancereadabilitybymakingitcleartoreadersthatvariablesofsubtypescan 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(inthecaseoftheassignedvaluebeingaliteralvalue)orbytherun-timesystem(inthecase 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 asintegers

– Subrange types are implemented like the parenttypes

 – code inserted (by the compiler) to restrict assignments to subrangevariables

Arrays and Indices

– Specificelementsofanarrayarereferencedbymeansofatwo-levelsyntacticmechanism, 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 shownas

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 subscriptvalue ranges are sometimes dynamicallybound.

– There are five categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage isallocated.

– A**static array**isoneinwhichthesubscript rangesarestaticallyboundandstorageallocation is static (done before runtime).

- The **advantage** of static arrays is efficiency: No dynamic allocation or deallocation isrequired.
- The **disadvantage i**s that the storage for the array is fixed for the entire execution time of theprogram.

– A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time duringexecution.

- The **advantage** of fixed stack-dynamic arrays over static arrays is space efficiency
- The **disadvantage** is the required allocation and deallocationtime

– 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 thevariable.

- The**advantage**ofstack-dynamicarraysoverstaticandfixedstack-dynamicarrays isflexibility

– 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 isallocated

- The **advantage** of fixed heap-dynamic arrays is flexibility—the array‘s size always fits theproblem.
- 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‘slifetime.

  – The **advantage** of heap-dynamic arrays over the others isflexibility:

  – The **disadvantage** is that allocation and deallocation take longer and may happen many times during execution of theprogram.

Array Initialization

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

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

In the C declaration

intlist [] = {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 themintheorderinwhichtheyaretobestored,orbydirectlyassigningthemtoanindexposition 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 multidimensioned 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 runtime.

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 equalnumberofvaluescalled**keys**.Inthecaseofnon-associativearrays,theindicesneverneed to be stored (because of their regularity). In an associative array, however, the user-definedkeys mustbestoredinthestructure.Soeachelementofanassociativearrayisinfactapairofentities, 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 theirelements.

Structure and Operations

InPerl,associativearraysarecalled**hashes**,becauseintheimplementationtheirelements arestoredandretrievedwithhashfunctions.ThenamespaceforPerlhashesisdistinct:Everyhash variablenamemust beginwithapercentsign(%).Eachhashelement consistsoftwoparts:akey, whichisastring,andavalue,whichisascalar(number,string,orreference).Hashescanbesetto literal values with the assignment statement, asin

%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,gradepointaverage,andsoforth.Adatatypeforsuchacollectionmight useacharacter stringforthename,anintegerforthestudentnumber,afloatingpointforthegradepointaverage, and so forth. Records are designed for this kind ofneed.

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**,arenotreferencedbyindices.Instead,thefieldsarenamedwithidentifiers,andreferences 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 followingexample:

– EMPLOYEE-RECORD.

– EMPLOYEE-NAME.

05 FIRST PICTURE IS X(20).

05 MIDDLE PICTURE IS X(10).

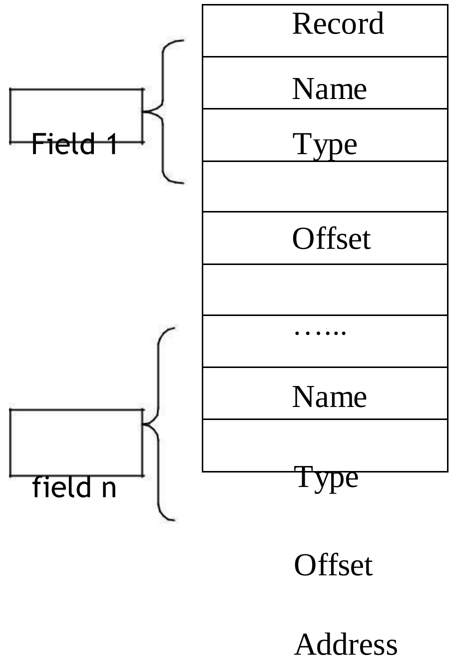
05 LAST PICTURE IS X(20).

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

Thefieldsofrecordsarestoredinadjacentmemorylocations.But becausethesizesofthe 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. Fieldaccessesareallhandledusingtheseoffsets.Thecompile-timedescriptorforarecordhasthe general form shown in Figure 6.7. Run-time descriptors for records areunnecessary



Union Types

A**union**isatypewhosevariablesmaystoredifferenttypevaluesatdifferenttimesduring programexecution.Asanexampleoftheneed forauniontype,consideratableofconstantsfora 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 canstore.

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 bedynamic.
- Should unions be embedded inrecords?

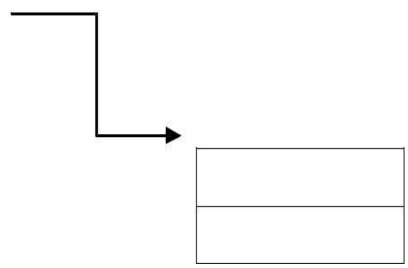
Ada Union Types

The Ada design for discriminated unions, which is based on that of its predecessor language, Pascal, allows the use[redacted]o 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;
whenFalse Sum :Float;
end case; **end record**;

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

Discriminated union
BOOLEAN
Offset
•
Address

True	•
False	•

count
integer

Sum

Float

Pointer and Reference Types

- A *pointer* is a variable whose value is anaddress
  - range of values that consists of memory addresses plus a special value,*nil*
- Provide the power of indirectaddressing
- Provide a way to manage dynamicmemory
- 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 singlenumber

Pointer Operations:

- Two fundamental operations: assignment anddereferencing
- Assignment is used to set a pointer variable's value to some usefuladdress
- Dereferencing yields the value stored at the location represented by the pointer'svalue
    - Dereferencing can be explicit orimplicit
    - C++ uses an explicit operation via*

j =*ptr

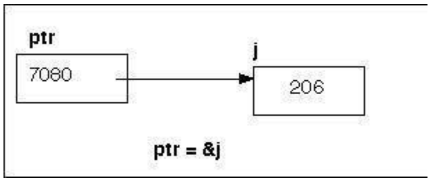
Pointers in C and C++:

- Extremely flexible but must be used withcare
- Pointers can point at any variable regardless of when it wasallocated
- Used for dynamic storage management andaddressing
- Pointer arithmetic ispossible
- Explicit dereferencing and address-ofoperators
- Domain type need not be fixed (**void***)
- void * can point to any type and can be type checked (cannot bede-referenced)

Pointer Operations Illustrated:

Pointer Operations Illustrated

Assignment ptr = &j

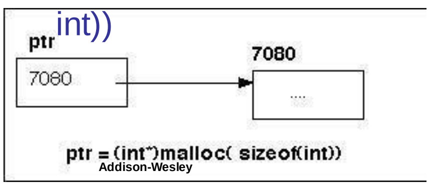


allocation

ptr = (int*)malloc(sizeof(int)

sizeof(

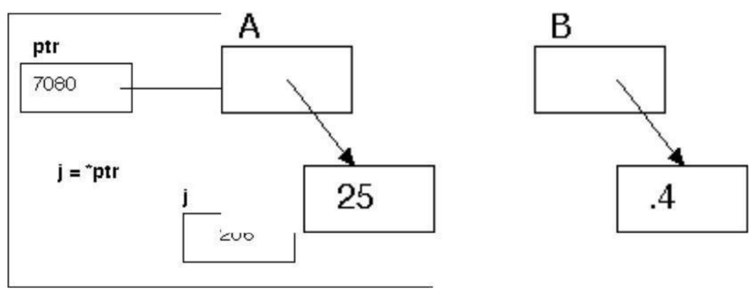
int))



Pointer Problems:
Dereferencing a pointer
j = *ptr

Pointer Problems

Dangling pointers(dangerous)

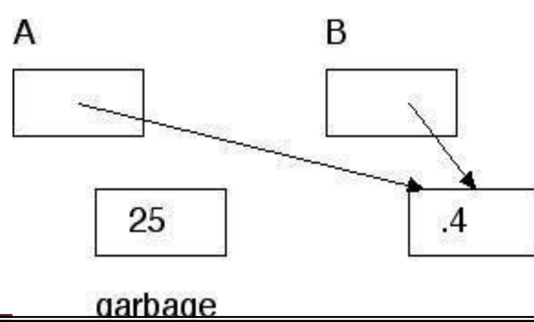


– A pointer points to a heap- dynamic variable that has been de-allocated

- Garbage

– An allocated heap-dynamic no

variable that is



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 formalparameters

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

- **Java extends C++**'s reference variables and allows them to replace pointersentirely

    –References refer to call instances

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

Evaluation of Pointers:

- Dangling pointers and dangling objects are problems as is heapmanagement
- Pointers are like goto's--they widen the range of cells that can be accessed by avariable
- Pointers or references are necessary for dynamic data structures--so we can't designa language withoutthem

Introduction to Names and variables:

- Imperative languages are abstractions of von Neumannarchitecture
- A machine consistsof
    - Memory - stores both data andinstructions
    - Processor - can modify the contents ofmemory
- Variables are used as an abstraction for memorycells
    - For primitive types correspondence isdirect

– For structured types (objects, arrays) things are morecomplicated

Names:

– **Why do we neednames?**

– needawaytorefertovariables,functions,user-definedtypes,labeledstatements, …

– **Design issues fornames:**

– Maximumlength?

– What characters areallowed?

– Are names casesensitive?

• C, C++, and Java names are casesensitive

• this is not true of otherlanguages

– Are special words reserved words orkeywords?

Length of Names:

– If too short, they cannot beconnotative

– Language examples:

– **FORTRAN** I: maximum6

– **COBOL**: maximum30

– **FORTRAN 90 and ANSI C:** maximum31

– **Ada and Java:** no limit, and all aresignificant

– **C++:** no limit, but implementers often imposeone

Keywords Vs Reserved Words:

– Words that have a special meaning in thelanguage

– A ***keyword*** is a word that is special only in certain contexts, e.g., inFortran

  • Real VarName(Real *is a data type followed with a name, therefore* Real *isa keyword)*

  • Real = 3.4 (*Real is avariable)*

– A ***reserved word*** is a special word that cannot be used as a user-definedname

  – most reserved words are alsokeywords

Variables:

– A variable is an abstraction of a memorycell

– Variables can be characterized as a sextuple ofattributes:

  – Name

  – Address

  – Value

  – Type

  – Lifetime

  – Scope

Variable Attributes:

– **Name -** not all variables havethem

– **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 asymbol
  - entity could be a variable or a function or even aclass
- **Binding time** is the time at which a binding takesplace.

Possible Binding Times:

- **Language design time** -- bind operator symbols tooperations
- **Language implementation time--** bind floating point type to arepresentation
- **Compile time --** bind a variable to a type in C orJava
- **Load time --** bind a FORTRAN 77 variable to a memory cell (or a C staticvariable)
- **Runtime --** bind a nonstatic local variable to a memorycell

Static and Dynamic Binding:

- A binding is ***static*** if it first occurs before run time and remains unchanged throughout programexecution.
- Abindingis ***dynamic***ifitfirstoccursduringexecutionorcanchangeduringexecutionof theprogram

Type Binding:

- **How is a typespecified?**
  - An ***explicit declaration*** is a program statement used for declaring the types of variables
  - An***implicitdeclaration***isadefaultmechanismforspecifyingtypesofvariables(the first appearance of the variable in theprogram)

- **When does the binding takeplace?**

- 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 subprogramsand assignments

- ***Type checking*** is the activity of ensuring that the operands of an operator are of compatibletypes

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 inappropriatetype

When is type checkingdone?

- 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 alwaysdetected

    - **Advantage**: allows the detection of misuse of variables that result in typeerrors

How strongly typed?

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

- **Pasca**l has variantrecords

- **C andC++**

    - parameter type checking can beavoided

    - unions are not type checked

– **Ada** isalmost

- **UNCHECKED CONVERSION** isloophole

– **Java** issimilar

Strong Typing:

– Advantage of strong typing: allows the detection of the misuses of variables that result in typeerrors

– Languageexamples:

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

– Pascal is not: variantrecords

– C and C++ are not: parameter type checking can be avoided; unions are not typechecked

- Ada is, almost (**UNCHECKED CONVERSION** is loophole) (Java issimilar)
- 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 stillfar less effective than that ofAda

Type Compatibility

- Our concern is primarily for structured types
- Def:Nametypecompatibilitymeansthetwovariableshavecompatibletypesiftheyarein either the same declaration or in declarations that use the same typename

– Easy to implement but highlyrestrictive:

– Subranges of integer types are not compatible with integertypes

– Formal parameters must be the same type as their corresponding actual

parameters (Pascal)

– Structuretypecompatibilitymeansthattwovariableshavecompatibletypesiftheirtypes have identicalstructures

– More flexible, but harder toimplement

- Consider the problem of two structuredtypes:

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 aredifferent?

(e.g. [1..10] and[0..9])

- Are two enumeration types compatible if their components are spelleddifferently?

- With structural type compatibility, you cannot differentiate between types of thesame structure (e.g. different units of speed, bothfloat)

Named Constants

Def: A named constant is a variable that is bound to a value only when it is bound to storage

- Advantages: readability andmodifiability
- Used to parameterizeprograms

- 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 typingconsiderably
  - **C and C++** allow both **widening and narrowingcoercions**
  - **Java** allows only **wideningcoercions**
    - **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 offlexibility
- 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 memorycell
  - **Allocation -** getting a cell from some pool of availablecells
  - **Deallocation -** putting a cell back into thepool
- Dependingonthelanguage,allocationcanbeeithercontrolledbytheprogrammerordone automatically

Categories of Variables by Lifetimes:

- **Static--**lifetime is same as that of theprogram
- **Stack-dynamic**--lifetime is duration ofsubprogram
- ***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 assignmentstatements

Variable Scope:

- The ***scope*** **of a variable** is the range of statements over which it is**visible**
- The***nonlocalvariables***ofaprogramunitarethosethatare**visible**but **notdeclared**there
- The scope rules of a language determine how references to names are associatedwith variables
- Scope and lifetime are sometimes closely related, but are differentconcepts

Static Scope:

- The **scope of a variable** can be determined from the **program text**
- Toconnect anamereferencetoavariable,you(orthecompiler)mustfindthedeclaration
- **Search process**: search declarations, first locally, then in increasingly larger enclosing scopes,untiloneisfoundforthegivenname**Enclosingstaticscopes**(toaspecificscope) are **called its static ancestors**; the nearest static ancestor is called a staticparent

Scope and Shadowed Variables:

- Variables can be hidden from a unit by having a "**closer**" variable with the samename
- **C++, Java and Ada** allow access to some of these "hidden" variables
  - **In Ada**:**unit.name**
  - **In C++: class_name::name** or **::name** forglobals

– **In Java**: **this.name** for variables declared at classlevel

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 inB

– **Solutions:**

  – Put D in B (but then C can no longer call it and D cannot access A'svariables)

  – Move the data from B that D needs to MAIN (but then all procedures canaccess them)

– Same problem for procedureaccess

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 selectedvariablesScope

- **Staticscoping**
  - Reference to x is to MAIN'sx
- **Dynamicscoping**
  - Reference to x is to SUB1'sx
- **Evaluation of DynamicScoping:**
  - **Advantage**:convenience
  - **Disadvantage:** poorreadability

Referencing Environments:

- The*referencingenvironment*ofastatementisthecollectionofallnamesthatarevisiblein the statement
- In a **static-scoped language**, it is the local variables plus all of the visible variables in all of the enclosingscopes
  - 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 yetterminate

Expressions and Statements Arithmetic Expressions:

- Arithmetic evaluation was one of the motivations forcomputers
- In programming languages, arithmetic expressions consist of operators,operands, parentheses, and functioncalls.
- An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has threeoperands.
- In most programming languages, binary operators are **infix**, which means they appear between theiroperands.
- 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 arithmeticcomputation

- Animplementationofsuchacomputation mustcausetwoactions:fetchingtheoperands, usually from memory, and executing arithmetic operations on thoseoperands.

- Arithmetic expressions consistof

- operators

- operands

  - parentheses

  - functioncalls

Issues for Arithmetic Expressions:

- operator precedencerules

- operator associativelyrules

- order of operandevaluation

- operand evaluation sideeffects

- operatoroverloading

- mode mixingexpressions

Operators:

A unary operator has one operand

- unary -,!

- A binary operator has twooperands

  - +, -, *, /,%

- A ternary operator hasthreeoperands -?:

Conditional Expressions:

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

- Anexample:

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

- Evaluates as if writtenlike

if (count == 0) average = 0 else average = sum /count

Operator Precedence Rules:

- *Precedencerules*definetheorderinwhich—adjacent‖operatorsofdifferentprecedence levels are evaluated
- Typical precedencelevels
  - parentheses
  - unaryoperators
  - ** (if the language supportsit)

Operator Associatively Rules:

- A*ssociativity rules* define the order in which adjacent operators with thesame precedence level areevaluated

- Typical associativityrules

  - Left to right for arithmeticoperators

  - exponentiation (** or ^) is right toleft

  - Sometimes unary operators associate right to left (e.g., inFORTRAN)

- APL is different; all operators have equal precedence and all operators associate rightto left

- Precedence and associativity rules can be overriden withparentheses

Operand Evaluation Order:

- Variables: fetch the value frommemory

- Constants: sometimes a fetch from memory; sometimes the constant is in themachine languageinstruction

- Parenthesized expressions: evaluate all operands and operatorsfirst

Potentials for Side Effects:

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

- Problem with functional sideeffects:

  - 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 theproblem

- Write the language definition to disallow functional sideeffects
  - No two-way parameters infunctions
  - No non-local references infunctions
  - Advantage: itworks!
  - Disadvantage: inflexibility of two-way parameters andnon-local references
- Write the language definition to demand that operand evaluation order befixed
  - Disadvantage: limits some compileroptimizations

Overloaded Operators:

– Use of an operator for more than one purpose is called *operatoroverloading*

– Some are common (e.g., + for intandfloat)

– Some are potential trouble (e.g., * in C andC++)

– Loss of compiler error detection (omission of an operand should be adetectable error)

– Some loss ofreadability

– Can be avoided by introduction of new symbols (e.g., Pascal‘s div forinteger division)

TypeConversions

A ***narrowingconversion*** convertsanobjecttoatypethatdoesnotincludeallofthevalues of the originaltype

- e.g., float toint

- A ***wideningconversion*** convertsanobjecttoatypethatcanincludeatleast approximations to all of the values of the originaltype

  - e.g., inttofloat

Coercion:

- A ***mixed-mode expression*** is one that has operands of differenttypes

- A ***coercion*** **is an implicit typeconversion**

- **Disadvantage** ofcoercions:

  - They decrease in the type error detection ability of thecompiler

- In most languages, widening conversions are allowed to happenimplicitly

- In Ada, there are virtually no coercions inexpressions

Casting:

- Explicit TypeConversions

- Called *casting* in C-basedlanguage

- 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-timesystem

Relational Operators:

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

Boolean Operators:

- Operands are Boolean and the result isBoolean
- **Exampleo**perators

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

No Boolean Type in C:

- C has no Booleantype
- it uses inttype with 0 for false and nonzero fortrue

Consequence

- a < b < c is a legalexpression
- the result is not what you might expect:

Left operator is evaluated, producing 0 or1

- 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/oroperators

  – Example: (13*a) * (b/13–1)

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

– Usually used for logicaloperators

– Problem with non-short-circuitevaluation

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** inta, b; float c; c = a /b;

– In **Java**, only widening assignment coercions aredone

– In **Ada,** there is no assignmentcoercion

Assignment Statements:

The general syntax

<target_var><assign_operator><expression>

- ➢ The assignmentoperator

– **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 ofassignment

- – 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 anddecrement operations withassignment

- – These have sideeffects

- – 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 anExpression:

- – In C, C++, and Java, the assignment statement produces a result and can be usedas operands

- – **Anexample:**

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 theissue

Oneimportantresult: Itwasproventhatallalgorithmsrepresentedbyflowchartscanbe coded with only two-way selection and pretest logicalloops

Control Structure

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

– Designquestion

–Should a control structure have multiple entries?

Selection Statements

– A *selection statement* provides the means of choosing between two or more pathsof execution

– Two general categories:

– Two-wayselectors

– Multiple-wayselectors

Two-Way Selection Statements

General form:

If control_expression then clause else clause

Design Issues:

- What is the form and type of the controlexpression?
- How are the **then** and **else** clauses specified?
- How should the meaning of nested selectors bespecified?

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 inparentheses
- 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 beBoolean

Clause Form

- In many contemporary languages, the then and else clauses can be single statementsor compoundstatements
- In Perl, all clauses must be delimited by braces (they must becompound)
- In Fortran 95, Ada, and Ruby, clauses are statementsequences
- Python uses indentation to define clauses if x >y :
  x =y

  print "case 1"

Nesting Selectors

- **Javaexample**

```
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++, andC#

– Perl requires that all then and else clauses to becompound

– Statement sequences as clauses: Ruby if sum == 0then

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 statementgroups

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++,andJavaswitch(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 integertype

– Selectable segments can be statement sequences, blocks, or compoundstatements

– Any number of segments can be executed in one execution of the construct (there isno implicit branch at the end of selectablesegments)

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

Multiple-Way Selection: Examples

– **C#**

➢ DiffersfromCinthatithasastaticsemanticsrulethatdisallowstheimplicitexecutionof more than onesegment

➢ Each selectable segment must end with an unconditional branch (goto orbreak)

– **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 casestatement

Ada design choices:

– Expression can be any ordinaltype

– Segments can be single orcompound

– Only one segment can be executed per execution of theconstruct

– Unrepresented values are notallowed

Constant List Forms:

– A list ofconstants

– Caninclude:

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 orrecursion

– General design issues for iteration controlstatements:

– How is iterationcontrolled?

– Where is the control mechanism in theloop?

Counter-Controlled Loops

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

– **DesignIssues**:

- 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 95syntax

  DO label var = start, finish [,stepsize]

- Stepsize can be any value butzero
- Parameters can beexpressions

– **Designchoices**:

- _ Loop variable must be **INTEGER**
- _ Loop variable always has its last value
- _ 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
- _ 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 Dostatements

Ada

for var in [reverse] discrete_range loop ... end loop

• **Designchoices**:

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

– Loop variable does not exist outside theloop

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

– The discrete range is evaluated justonce

– Cannot branch into the loopbody

– C-basedlanguages

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

- The expressions can be whole statements, or even statement sequences, with the statements separated bycommas
- 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 infiniteloop

Design choices:

– There is no explicit loopvariable

– Everything can be changed in theloop

– The first expression is evaluated once, but the other two are evaluated with each iteration

C++ differs from C in twoways:

- The control expression can also beBoolean
- Theinitialexpressioncanincludevariabledefinitions(scopeisfromthedefinitiontothe end of the loopbody)
- Java andC#

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

Iterative Statements: Logically-Controlled Loops: Examples

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

  while (ctrl_expr) do loop body loop body while (ctrl_expr)

- JavaislikeCandC++,exceptthecontrolexpressionmustbeBoolean(andthebodycan only be entered at the beginning -- Java has no**goto**

Iterative Statements: Logically-Controlled Loops: Examples

- Ada has a pretest version, but noposttest
- FORTRAN 95 hasneither
- Perl and Ruby havetwo 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 theloop)
- Simple design for single loops (e.g.,break)
- Design issues for nestedloops
- Should the conditional be part of theexit?
- Should control be transferable out of more than oneloop?

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** inPerl)
- C, C++, and Python have an unlabeled control statement, **continue**, that skips the remainder of the current iteration, but does not exit theloop
- JavaandPerlhave labeledversionsof**continueIterativeStatements**:IterationBasedon DataStructures
- 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 isterminate
- C's**for**canbeusedtobuildauser-defined iterator:for(p=root;p==NULL;traverse(p)){}
- C#_s **foreach**statement iterates on the elements of arrays and othercollections:

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 bedisplayed

– Perl has a built-in iterator for arrays and hashes, **foreach UnconditionalBranching**

– Transfers execution control to a specified place in theprogram

– Represented one of the most heated debates in 1960_s and1970_s

– Well-known mechanism: gotostatement

– Major concern:Readability

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

– C# offers goto statement (can be used in switchstatements)

– Loop exit statements are restricted and somewhat camouflagedgoto_s

Guarded Commands

- Designed byDijkstra
- Purpose: to support a new programming methodology that supportedverification (correctness) duringdevelopment
- Basis for two linguistic mechanisms for concurrent programming (in CSP andAda)

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 isreached,

–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

1. Evaluate all Booleanexpressions
2. If more than one are true, choose one non-deterministically; then start loopagain
3. If none are true, exitloop

Guarded Commands: Rationale

- Connection between control statements and program verification isintimate
- Verification is impossible with gotostatements
- Verification is possible with only selection and logical pretestloops
- Verification is relatively simple with only guardedcommands

UNIT-3

BasicDefinitions:

- Asubprogramdefinitionisa descriptionofthe actions ofthe subprogramabstraction
- A subprogram call is an explicit request that the subprogram beexecuted
- A subprogram header is the first line of the definition, including the name, the kindof
- subprogram, and the formalparameters
- The parameter profile of a subprogram is the number, order, and types of itsparameters
- 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 thesubprogram
- 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 callStatement

Actual/Formal Param Correspondence Two basic choices:

- Positional
- Keyword
- Sort (List =>A, Length =>N);
- For namedassociation:
- **Advantage**: order isirrelevant

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-definedoperators

Design Issues for Subprograms

– What parameter passing methods areprovided?

– Are parameter typeschecked?

– Are local variables static ordynamic?

– What is the referencing environment of a passedsubprogram?

– Are parameter types in passed subprogramschecked?

– Can subprogram definitions benested?

– Can subprograms beoverloaded?

– Are subprograms allowed to begeneric?

– Is separate/independent compilationsupported?

– ReferencingEnvironments

– If local variables arestack-dynamic:

Advantages:

- Support forrecursion
- Storage for locals is shared among somesubprograms

Disadvantages:

- Allocation/deallocationtime
- Indirectaddressing
- Subprograms cannot be historysensitive
- Static locals are theopposite

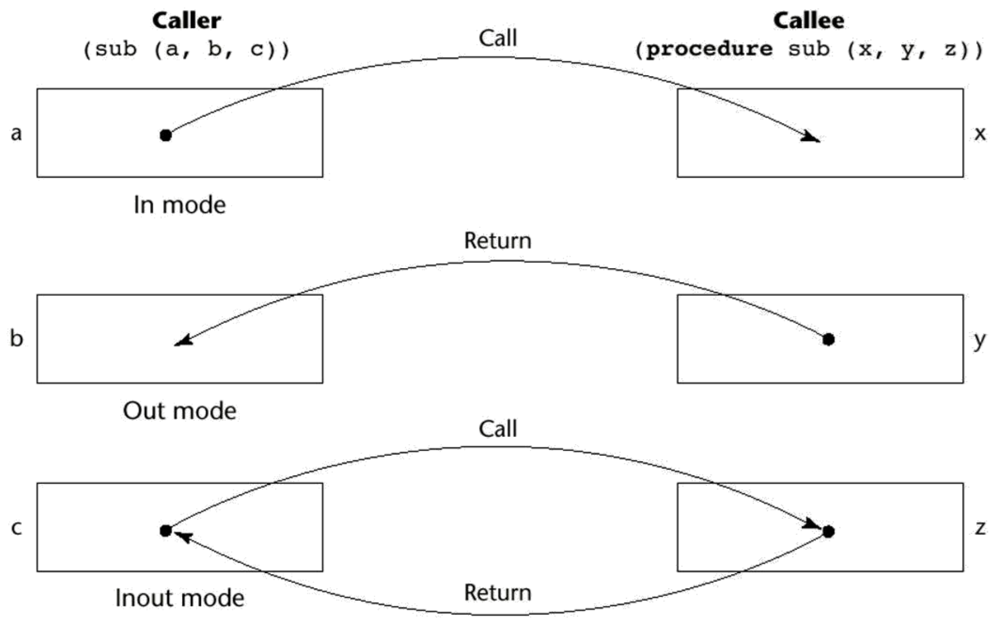
Parameters and Parameter Passing: Semantic Models: in mode, out mode, inout mode
Conceptual Models of Transfer:

○ 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

– inmode

– Either by physical move or accesspath

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**

– outmode

Disadvantages:

– If value is moved, time andspace

– 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 thereturn

Pass-By-Value-Result:

sub1(i, a[i]);

Pass-By-Name Example 2

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]);/*ifi=j*/CollisionbetweenformalsandglobalsRootcauseofallof these is: The called subprogram is provided wider access to non locals than isnecessary

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

Pass-By-NamemultiplemodesBytextualsubstitution,Formalsareboundtoanaccessmethodat thetimeofthecall,butactualbindingtoavalueoraddresstakesplaceatthetimeofareferenceor 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;
```

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 inefficientreferences

– Too tricky; hard to read andunderstand

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-valueisoptional
- ALGOLW:Pass-by-value-result,C:Pass-by-valuePascalandModula-2:Default is pass-by value;,pass-by-referenceisoptional

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

ALGOL60and mostofitsdescendantsusetheruntime stackValue—copyittothestack; references are indirect to the stack Result—sum, Reference—regardless of form, put the address in thestack

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 resultsProgramswithsucherrorsare―erroneous‖

Multidimensional Arrays as Params

If a multidimensional array is passed to a subprogram and the subprogram is separately compiled,thecompilerneedsto knowthedeclaredsizeofthatarraytobuild thestoragemapping 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 ortwo-way

– These two are in conflict with oneanother!

– Good programming => limited accessto

– variables, which means one-way wheneverpossible

– Efficiency => pass by reference is fastest way to pass structures of significant size Also, functions should not allow referenceParameters

Subprograms As Parameters:Issues

Are parameter typeschecked?

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

– A generic or polymorphic subprogram is one that takes parameters of different typeson different activations Overloaded subprograms provide ad hocpolymorphism

– A subprogram that takes a generic parameter that is used in a type expression thatdescribes the type of the parameters of the subprogram provides parametricpolymorphism

– See Ada generic and C++ template examples intext

– Independent compilation is compilation of some of the units of a program separately fromthe rest of the program, without the benefit of interface information

– Separatecompilationiscompilationofsomeoftheunitsofaprogramseparatelyfromtherest of the program, using interface information to check the correctness of the interface between the twoparts

Language Examples:

– FORTRAN II to FORTRAN 77:independent

– FORTRAN 90, Ada, Modula-2, C++:separate

– Pascal: allowsneither

Functions Design Issues:

– Are side effectsallowed?

– Two-way parameters (Ada does notallow)

– Nonlocal reference (allallow)

– What types of return values areallowed?

– FORTRAN, Pascal, Modula-2: only simpletypes

– **C**: any type except functions andarrays

– **Ada:** any type (but subprograms are nottypes)

– **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 notnested

– Globals are created by external declarations (they are simply defined outside anyfunction)

– Access is by either implicit or explicitdeclaration

– Declarations(notdefinitions) givetypestoexternallydefinedvariables(andsaytheyare definedelsewhere)

– External modules: Ada andModula-2:

– DynamicScope:

User-Defined Overloaded Operators

- Nearly all programming languages have overloaded operators
- Users can further overload operators in C++ and Ada not carried over intoJava)

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:

CoroutineisasubprogramthathasmultipleentriesandcontrolsthemitselfAlsocalled symmetric control A coroutine call is named a resume. The first resume of a coroutineis 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 notoverlapped

UNIT-4

Abstraction:

- The concept of abstraction is fundamental inprogramming
- Nearly all programming languages support process abstraction withsubprograms
- Nearly all programming languages designed since 1980 have supported dataabstraction with some kind of module

Encapsulation:

– **Originalmotivation**:

– Large programs have two special needs:
– Some means of organization, other than simply division into subprograms
– Some means of partial compilation (compilation units that are smaller than the whole program)

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

– These are called encapsulations

Examples of Encapsulation Mechanisms:

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

– 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:

Definition1: Therepresentationofandoperationsofobjectsofthetypearedefined ina single syntactic unit; also, other units can create objects of thetype.

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.

Concurrency can occur at four levels:

– **Machine instructionlevel**

– **High-level language statementlevel**

– **Unitlevel**

– **Programlevel**

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 morespecial-purpose processors for input and outputoperations

_ **Early 1960s -** Multiple complete processors, used for program-level concurrency

– **Mid-1960s -** Multiple partial processors, used for instruction-levelconcurrency

– **Single-Instruction Multiple-Data (SIMD) machine** The same instruction goes toallprocessors, each with different data - e.g., *vectorprocessors*

– **Multiple-Instruction Multiple-Data (MIMD)** machines, Independent processors that can be synchronized (unit-levelconcurrency)

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

Categories of Concurrency:

– **Physical concurrency -** Multiple independent processors (multiple threads ofcontrol)

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

- **Coroutines***provide only***quasiconcurrency**

Reasons to Study Concurrency:

1. It involves a new way of designing software that can be very useful--manyreal-world situation involveconcurrency

2. Computers capable of physical concurrency are now widelyused

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 inthat:

- A task may be implicitlystarted
- When a program unit starts the execution of a task, it is not necessarilysuspended
- When a task's execution is completed, control may not return to thecaller

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 bethrough:**

- **Shared nonlocal variables**
- **Parameters**
- **Message passing**

– **Kinds ofsynchronization**:

- **Cooperation**

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

e.g., the producer-consumer problem

- **Competition**

When two or more tasks must use some resource that cannot be simultaneously used **.,** a shared counter. A problem because operations are not atomic

- Competition is usually provided by **mutually exclusive access** (methods are discussedlater
- Providing synchronization requires a mechanism for delaying taskexecution
- Task execution control is maintained by a program called the scheduler, whichmaps task execution onto availableprocessors

Tasks can be in one of several different execution states:

- New - created but not yetstarted
- Runnable or ready - ready to run but not currently running (noavailable processor)
- Running
- Blocked - has been running, but cannot not continue (usually waiting for some event to occur)
- Dead - no longer active in anysense

Liveness is a characteristic that a program unit may or may not have

- In sequential code, it means the unit will eventually complete itsexecution
- In a concurrent environment, a task can easily lose itsliveness

If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

–**Design Issues forConcurrency***:*

_ How is cooperation synchronizationprovided?

_ How is competition synchronizationprovided?

_ How and when do tasks begin and end execution?

_ Are tasks statically or dynamically created?

Example: A buffer and some producers and some consumers

Techniqu*e:* Attach two SIGNAL objects to the buffer, one for full spots and one for empty spot
Methods of Providing Synchronization:

- Semaphores
- Monitors
- MessagePassing

– 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 datastructures

– Semaphoreshaveonlytwooperations,waitandrelease(originallycalledPandVbyDijkstra)
– 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

–Thesemaphorecountersareusedtostorethenumbersofemptyspotsandfullspots 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 emptyspots

– When DEPOSIT is finished, it must increment the counter of full spots

FETCH must first check full spots to see if there is avalue

- If there is a full spot, the counter offull 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 fullspots

– When FETCH is finished, it increments the counter of emptyspots

– TheoperationsofFETCHandDEPOSITonthesemaphoresareaccomplishedthrough two semaphore operations named wait and release wait(aSemaphore) if a Semaphore‘s counter > 0 then Decrement aSemaphore‘s counter else

PutthecallerinaSemaphore‘squeueAttempttotransfercontroltosomereadytask(Ifthe task ready queue is empty, deadlock occurs)end

release(aSemaphore)
if aSemaphore‘s queue is empty then Increment aSemaphore‘scounter

else

Putthecallingtaskinthe taskreadyqueueTransfercontroltoataskfromaSemaphore‘squeue end

– **Competition Synchronization withSemaphores:**

A third semaphore, named access, is used to control access (competitionsynchronization)
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:

– Misuse of semaphores can cause failures in cooperationsynchronization
e.g., the buffer will overflow if the wait of full spots is left out

– Misuseofsemaphorescancausefailuresincompetitionsynchronizatione.g., Theprogram will deadlock if the release of access is leftout.

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 themonitor
- Multiple calls arequeued

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 torelease)

➢ delay takes a queue type parameter; it puts the process that calls it in the specified queueand removes its exclusive access rights to the monitor‘s datastructure

➢ Differs from send because delay always blocks thecaller

➢ continuetakesaqueuetypeparameter;itdisconnectsthecallerfromthemonitor,thusfreeing the monitor for use by anotherprocess.

➢ It also takes a process from the parameter queue (if the queue isn‘t empty) and starts it, Differsfromreleasebecauseitalwayshassomeeffect(releasedoesnothingifthequeueisempty)

Java Threads

The concurrent units in Java are methods named run

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

– 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 ofthreads
- The yield is a request from the running thread to voluntarily surrender theprocessor
- The sleep method can be used by the caller of the method to block thethread
- The join method is used to force a method to delay its execution until the runmethod of another thread has completed itsexecution

Thread Priorities

– A thread_s default priority is the same as the thread that createit.
– If main creates a thread, its default priority isNORM_PRIORITY

–Threads defined two other priority constants, MAX_PRIORITY andMIN_PRIORITY

–The priority of a thread can be changed with the methodssetPriority

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 inheritthem

–The wait method must be called in aloop

–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 waitlist

Java's Thread Evaluation

–Java_s support for concurrency is relatively simple buteffective

–Not as powerful as Ada_stasks

C# Threads

— Loosely based on Java but there are significantdifferences
–Basic threadoperations

– Any method can run in its ownthread

– A thread is created by creating a Threadobject

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

– A thread can be made to wait for another thread to finish withJoin

– A thread can be suspended withSleep

– A thread can be terminated withAbort

Synchronizing Threads

— Three ways to synchronize C#threads
– The Interlockedclass
– Usedwhentheonlyoperationsthatneedtobesynchronizedareincrementingordecrementing of aninteger

– The lockstatement

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

– The Monitorclass

– Provides four methods that can be used to providemore

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 exceptionhandler
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:**

_ Send an auxiliary parameter or use the return value to indicate the return status of a Subprogram

- e.g., C standard libraryfunctions

_ Pass a label parameter to all subprograms (error return is to the passed label)

- e.g.,FORTRAN

_ Pass an exception handling subprogram to all subprograms

Advantages of Built-in Exception Handling:

– Error detection code is tedious to write and it clutters theprogram

– Exception propagation allows a high level of reuse of exception handlingcode

Design Issues for Exception Handling:

– How and where are exception handlers specified and what is theirscope?

– How is an exception occurrence bound to an exceptionhandler?

– Where does execution continue, if at all, after an exception handler completes its execution?

– How are user-defined exceptions specified?

– Should there be default exception handlers for programs that do not provide theirown?

– Can built-in exceptions be explicitlyraised?

– Are hardware-detectable errors treated as exceptions that can behandled?

– Are there any built-inexceptions?

– How can exceptions be disabled, if atall?

PL/I Exception Handling

- Exception handlerform:

  EX: ON condition [SNAP] BEGIN; ... END;

  - condition is the name of the associatedexception
  - SNAP causes the production of a dynamic trace to the point of theexception
  - Binding exceptions tohandlers

It is dynamic--binding is to the most recently executed ON statement

- Continuation
  - Some built-in exceptions return control to the statement where the exceptionwas raised
  - Others cause programtermination
  - User-defined exceptions can be designed to go to any place in the program that is labeled

- Other designchoices:

  – User-defined exceptions are defined with: CONDITIONexception_name

– Exceptions can be explicitly raised with: SIGNAL CONDITION(exception_name)
– Built-in exceptions were designed into threecategories:

- Those that are enabled by default but could be disabled by usercode
- Those that are disabled by default but could be enabled by usercode
- Those that are always enabled

Evaluation

- The design is powerful and flexible, but has the followingproblems:

– Dynamic binding of exceptions to handler makes programs difficult to write and toread

– The continuation rules are difficult to implement and they make programs hard toread

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 SLDresolution

– Mostly developed in fifth generation computer systemsproject

– Specially designed for theorem proof and artificial intelligence butallows general purposecomputation.

– 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 consistof
  - o [a-zA-Z0-9]*
  - a aDAM a12
- 2 Strings consisting of onlypunctuation

- *** .+. .<.>.

- 3 Any string enclosed in single quotes (like an arbitrarystring)
  - ‘ADAM‘ ‘Onur Sehitoglu‘‘2 * 4 <6‘
- Numbers
  - 1234 12.32 12.23e-10

Variables:

- Strings with starting with a capital letter or and consistof

- [a-zA-Z0-9]*

- Adam adamA093
- is the universal match symbol. Notvariable

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](asciiintegervalues)usedisplay(Term).toseeactualstructureof theterm

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. ifS isavariable,SisinstantiatedasT,ifitiscompatiblewithcurrentconstraintstore(S is instantiated to another term, they areunified)
3. if S and T are structures, successful if: head of S = head of T they have samearity unification of all corressponding terms aresuccessful.

S: list of structures, P current constraint store

s2S,arity(s):numberofargumentsofstructure,s2S,head(s):headatomofthe structure,

s 2 S, argi (s): ithargument term of the structure, p _ P: p is consistent with currentconstraintstore.S_T;P=(S,T2A_S,T2N)^S=T!true‘sS2V^S_T|=P! true;S_T ^PT2V^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**UnificationExample:**X=a !pwithX=aa(X,3)=a(X,3,2)!×a(X,3)=b(X,3)!×a(X,3) =a(3,X)!pwithX=3a(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]) :- prefixof(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 ordercontrol
- The closed-worldassumption
- The negationproblem
- Intrinsiclimitations
- Applications of LogicProgramming
- Relational database managementsystems
- Expertsystems
- Natural languageprocessing

UNIT-5

Functional programming languages

Functional Programming Languages:

- Thedesignoftheimperativelanguagesisbaseddirectlyonthevon Neumannarchitecture
- Efficiency is the primary concern, rather than the suitability of the language forsoftware development.
- The design of the functional languages is based on mathematical functions
- Asolidtheoreticalbasisthatisalsoclosertotheuser,butrelativelyunconcernedwiththe architecture of the machines on which programs willrun

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. **FunctionComposition**:

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:**

Afunctionalformthattakesalist offunctionsasparametersandyieldsalistoftheresults 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 andlists

2. List form: parenthesized collections of sub lists and/oratoms 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 lateruse
- Management of variables is a constant concern and source of complexity forimperative programming
- In an FPL, variables are not necessary, as is the case inmathematics
- In an FPL, the evaluation of a function always produces the same result given thesame parameters
- This is called referentialtransparency

A Bit of LISP:

– Originally, LISP was a type less language. There were only two data types, atom andlist

– LISP lists are stored internally as single-linkedlists

– Lambdanotationisusedtospecifyfunctionsandfunctiondefinitions, functionapplications, and data all have the sameform.

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 versionthan the contemporary dialects of LISP, Uses only staticscoping

– Functions are first-class entities, They can be the values of expressions and elements oflists, They can be assigned to variables and passed asparameters

Primitive Functions:

_ **Arithmetic**: +, -, *, /, ABS, SQRT **Ex:** (+ 5 2) yields 7

– **QUOTE:** -takes one parameter; returns the parameter withoutevaluation

- **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 notappropriate
- **QUOTE** can be abbreviated with the apostrophe prefix operator e.g., '**(A B)** is equivalent to **(QUOTE (AB))**
- **CAR** takes a list parameter; returns the first element of thatlist

  e.g., (CAR '(A B C)) yields **A (CAR '((A B) C D))** yields **(A B)**
- **CDR**takesalist parameter;returnsthe list afterremovingitsfirstelemente.g.,(CDR'(A B C)) yields (B C)

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

- 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 itsresult

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

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

– Predicate Functions: (#T and () are true andfalse)

_ EQ?takestwosymbolicparameters;itreturns#Tifbothparametersareatomsandthetwo are thesame

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

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

- NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise ()
  Note that NULL? returns #T if the parameter is ()

- Numeric PredicateFunctions

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

Output Utility Functions: (DISPLAY expression) (NEWLINE)

- Lambda Expressions
- Form is based on notatione.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)CD))

- A Function for Constructing Functions DEFINE - Twoforms:
- To bind a symbol to anexpression

EX:

```
(DEFINE    pi    3.141593)
(DEFINE two_pi (* 2 pi))
```

To bind names to lambda expressions

EX:

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

Evaluation process (for normal functions):

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- Thevalueofthelast expressioninthebodyisthevalueofthefunction(Specialformsuse a different evaluationprocess)

Control Flow:

Selection- the special form, IF (IF predicate then_expelse_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
- 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 datatypes
- Includes lists and listoperations
- 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

Applications of Functional Languages:

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

Scheme is used to teach introductory

– programming at a significant number ofuniversities

Comparing Functional and Imperative Languages Imperative Languages:

- Efficient execution o Complex semantics o Complexsyntax

- Concurrency is programmerdesigned

– **FunctionalLanguages:**

- Simplesemantics
- Simplesyntax
- Inefficientexecution
- Programs can automatically be madeconcurrent

Scripting languages

Pragmatics

– *Scripting* is a paradigm characterizedby:

- use of scripts to glue subsystemstogether;
- rapid development and evolution ofscripts;
- modest efficiencyrequirements;
- very high-level functionality in application-specificareas.

– A software system often consists of a number of subsystems controlledor connected by ascript.

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

COMMON CHARACTERISTICS OF SCRIPTING LANGUAGES

– Both batch and interactiveuse

– Economy ofexpressions

– Lack of declaration; simple scopingrules

– Flexible dynamictyping

– Easy access to otherprograms

- High level datatypes
- Glue other programstogether
- Extensive text processingcapabilities
- Portable across windows, unix,mac

PYTHON

- PYTHON was designed in the early 1990s by Guido vanRossum.
- 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 functionormoduletotheinterpreter,eithertoperformcriticaloperationsatmaximumspeedofto link python programs to libraries that may only be available in binaryform.

Python has following characteristics.

- Easy to learn and program and is objectoriented.
- Rapid application development
- Readability is better
- It can work with other languages such as C,C++ andFortran
- Powerfulinterpreter
- Extensive modules support isavailable

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 usedinstead.

- Its boolean values (named False and True) are just smallintegers.

- PYTHON has a rich repertoire of composite types: tuples, strings,lists, dictionaries, andobjects.

Variables, storage, and control

- PYTHON supports global and localvariables.

- Variables are not explicitly declared, simply initialized byassignment.

- PYTHON adopts reference semantics. This is especially significant formutable values, which can be selectivelyupdated.

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, conditional (if- butnotcase-) commands, iterative (while- and for-) commands, and exception-handlingcommands.

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 declareclasses
- Within a procedure we may initialize local variables and define localprocedures.
- Within a class we may initialize variable components and defineprocedures (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;

Proceduralabstraction

– PYTHON supports function procedures and properprocedures.

– The only difference is that a function procedure returns a value, while aproper procedure returnsnothing.

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

Python procedure
Eg :Defgcd (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

elifval> max: max = val return min, max

Data Abstraction

– PYTHON has three different constructs relevant to data abstraction: packages ,modules, andclasses

– Modulesandclassessupportencapsulation,using anamingconventiontodistinguish between public and private components.

– A Package is simply a group ofmodules

– A Module is a group of components that may be variables, procedures, andclasses

– AClassisagroupofcomponentsthatmaybeclassvariables,classmethods,andinstance 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. Otherwisethe procedure acts as a classmethod.

Separate Compilation

- PYTHON modules are compiledseparately.
- Each module must explicitly import every other module on which itdepends
- 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 namedprogram.pyc
- Compilation is completely automatic

- The PYTHON compiler does not reject code that refers to undeclared identifiers. Suchcode simply fails if and when it isexecuted
- The compiler will not reject code that might fail with a type error, nor even code that will certainly fail, suchas:

    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,andcryptography,multimedia,GUIs,operatingsystemservices, internet services,compilation, and soon.
- Unlikeolderscriptinglanguages,PYTHONdoesnothavebuilt-inhigh-levelstring processing or GUI support, so module library providesit