PRINCIPLES OF PROGRAMMING LANGUAGES

III B. Tech I semester (JNTUH-R15)

Ms. K. Radhika
Associate Professor

Ms. B. Jaya Vijaya
Assistant Professor

Mr. P. Sunil Kumar
Assistant Professor

UNIT-1

Preliminaries

Syntax and Semantics

CONCEPTS

Reasons for Studying Concepts of Programming Languages.

Programming Domains

Language Evaluation Criteria

Influences on Language Design

Language Categories

Language Design Trade-Offs

Implementation Methods

Programming Environments

CONCEPTS

Introduction to syntax and semantics

The General Problem of Describing Syntax

Formal Methods of Describing Syntax

Attribute Grammars

Describing the Meanings of Programs: Dynamic Semantics

❖ *Reasons for Studying Concepts of Programming Languages*

Increased ability to express ideas.

Improved background for choosing appropriate languages.

Increased ability to learn new languages.

Better understanding of significance of implementation.

Better use of languages that are already known.

Overall advancement of computing.

❖ *Programming Domains*

Scientific Applications

– Large numbers of floating point computations; use of arrays.

– Example:Fortran.

Business Applications

– Produce reports, use decimal numbers and characters.

– Example:COBOL.

Artificial intelligence

– Symbols rather than numbers manipulated; use of linked lists.

– Example:LISP.

❖ ***Programming Domains***

System programming

Need effieciency because of continous use.
Example:C

Web Software

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

❖ *Language Evaluation Criteria*

Readability:

The ease with which programs can be read and understood.

Writability:

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

Reliability:

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

Cost:

- The ultimate total cost.

❖ *Evaluation Criteria: Readability*

Overall simplicity

A manageable set of features and constructs.

Minimal feature multiplicity .

Minimal operator overloading.

Orthogonality

A relatively small set of primitive constructs can be combined in a relatively small number of ways

Every possible combination is legal

Data types

Adequate predefined data types.

❖ ***Evaluation Criteria:Readability***

Syntax considerations

-Identifier forms:flexible composition. -Special words and methods of forming compound statements.

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

❖ *Evaluation Criteria: Writability*

Simplicity and orthogonality

– Few constructs, a small number of primitives, a small set of rules for combining them.

Support for abstraction

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

Expressivity

– A set of relatively convenient ways of specifying operations.

– Strength and number of operators and predefined functions.

❖ *Evaluation Criteria: Reliability*

Type checking

– Testing for type errors.

Exception handling

– Intercept run-time errors and take corrective measures.

Aliasing

– Presence of two or more distinct referencing methods for the same memory location.

Readability and writability

– A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability.

❖ *Evaluation Criteria: Cost*

Training programmers to use the language

Writing programs (closeness to particular applications)

Compiling programs

Executing programs

Language implementation system: availability of free compilers

Reliability: poor reliability leads to high costs

Maintaining programs

Evaluation Criteria: Others

Portability

– The ease with which programs can be moved from one implementation to another.

Generality

– The applicability to a wide range of applications.

Well-definedness

– The completeness and precision of the language's official definition.

❖ *Influences on Language Design*

Computer Architecture

– Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture

Programming Methodologies

– New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

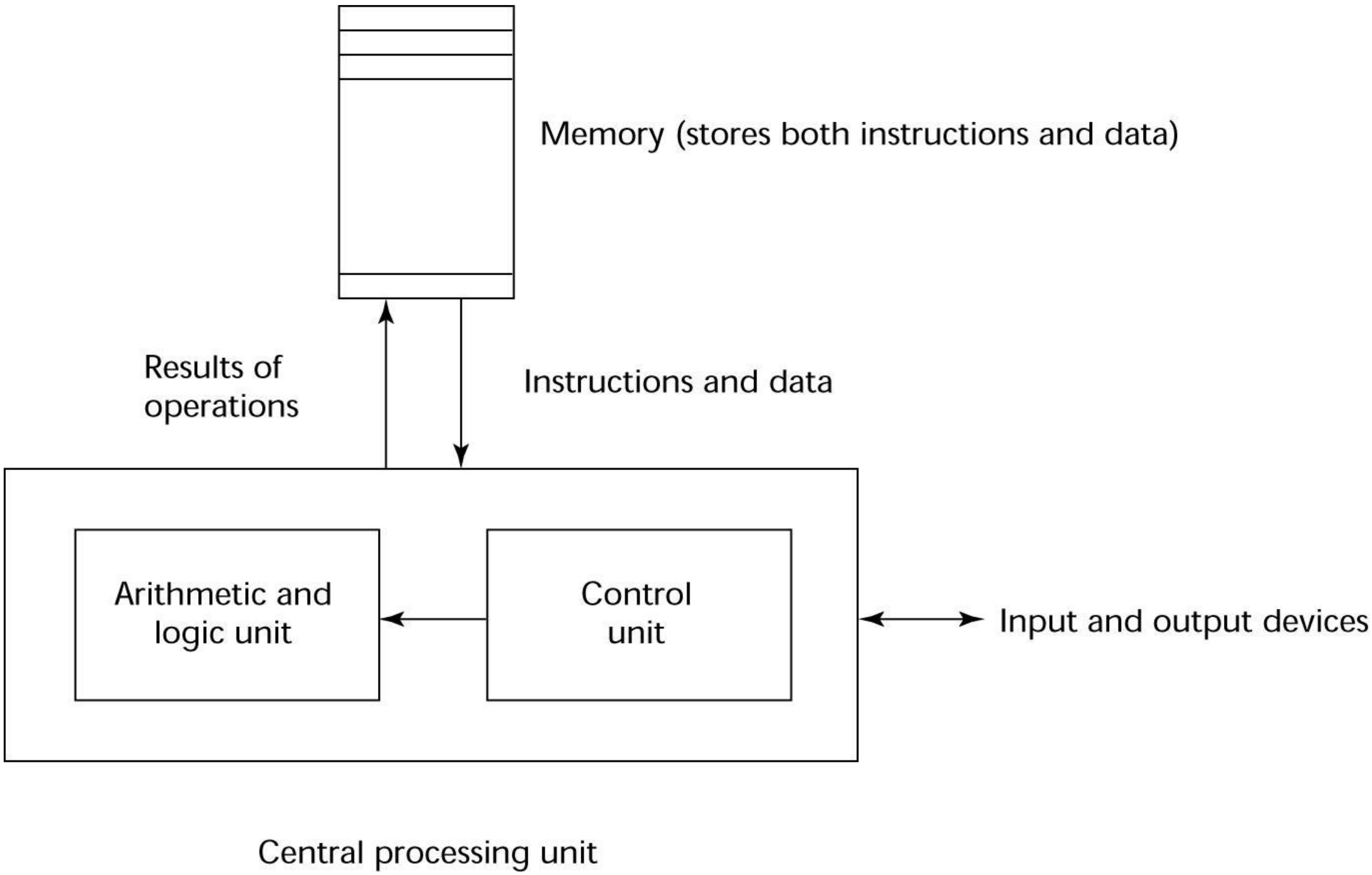
❖ *Computer Architecture Influence*

Well-known computer architecture: Von Neumann

Imperative languages, most dominant, because of von Neumann computers

- Data and programs stored in memory
- Memory is separate from CPU
- Instructions and data are piped from memory to CPU
- Basis for imperative languages
  - Variables model memory cells
  - Assignment statements model piping
  - Iteration is efficient

❖ *The Von Neumann Architecture*



❖ *The Von Neumann Architecture*

Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program
counter repeat forever
  fetch the instruction pointed by the
  counter increment the counter
  decode the instruction
  execute the instruction
end repeat
```

❖ *Programming Methodologies Influences*

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

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

- structured programming
- top-down design and step-wise refinement

Late 1970s: Process-oriented to data-oriented

- data abstraction

Middle 1980s: Object-oriented programming

- Data abstraction + inheritance + polymorphism

❖ *Language Categories*

Imperative

- Central features are variables, assignment statements, and iteration
- Include languages that support object-oriented programming
- Include scripting languages
- Include the visual languages
- Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++

Functional

- Main means of making computations is by applying functions to given parameters
- Examples: LISP, Scheme

Logic

- Rule-based (rules are specified in no particular order)
- Example: Prolog

Markup/programming hybrid

- Markup languages extended to support some programming
- Examples: JSTL, XSLT

❖ *Language Design Trade-Offs*

Reliability vs. cost of execution

– Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs

Readability vs. writability

Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

Writability (flexibility) vs. reliability

– Example: C++ pointers are powerful and very flexible but are unreliable

❖ *Implementation Methods*

Compilation

– Programs are translated into machine language

Pure Interpretation

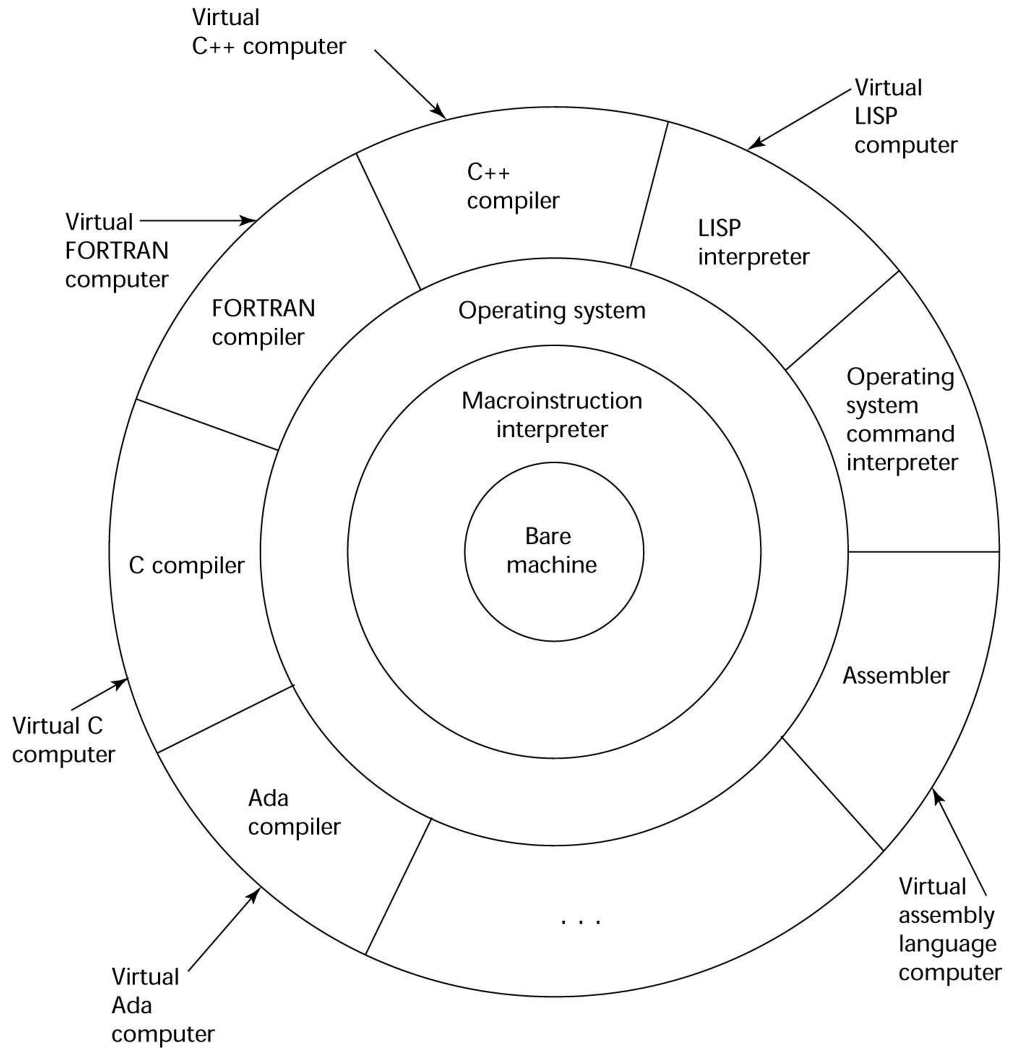
– Programs are interpreted by another program known as an interpreter

Hybrid Implementation Systems

– A compromise between compilers and pure interpreters

❖ *Layered View of Computer*

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



Compilation

Translate high-level program (source language) into machine code (machine language)

Slow translation, fast execution

Compilation process has several phases:

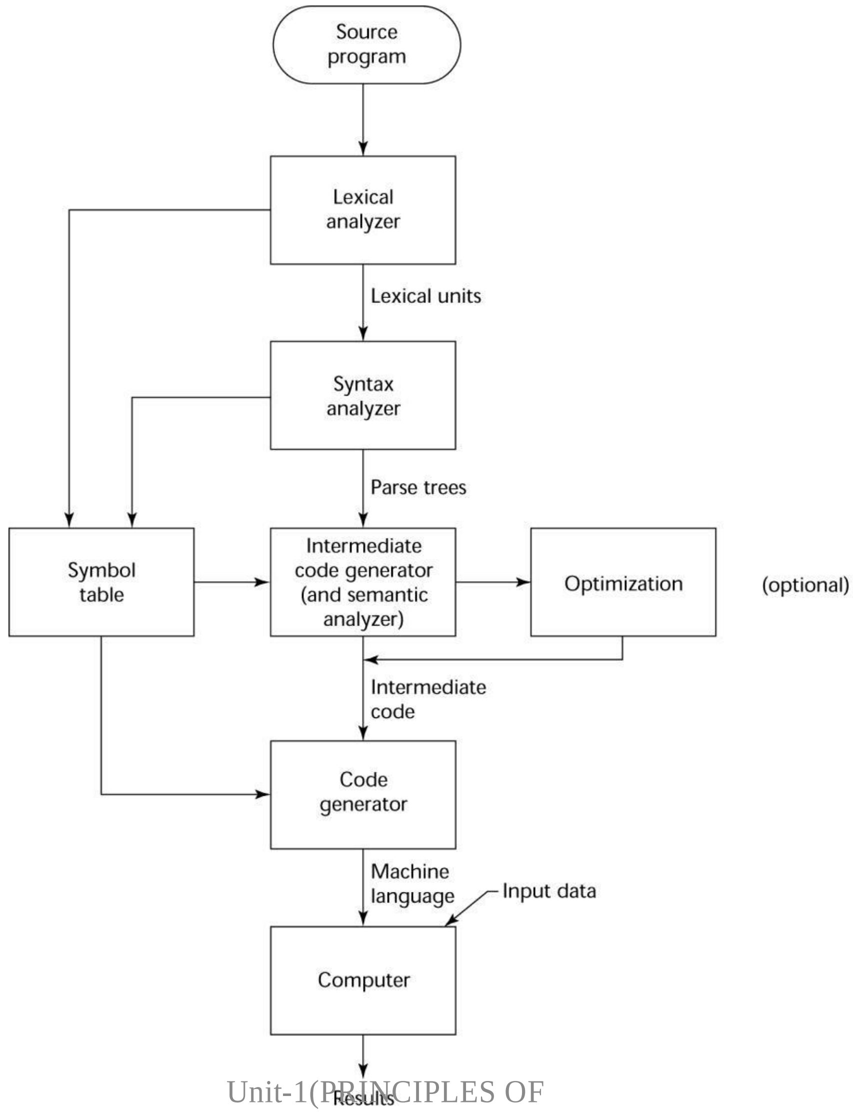
– lexical analysis: converts characters in the source program into lexical units

– syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program

– Semantics analysis: generate intermediate code

– code generation: machine code is generated

The Compilation Process



Additional Compilation Terminologies

Load module (executable image): the user and system code together

Linking and loading: the process of collecting system program units and linking them to a user program

Von Neumann Bottleneck

Connection speed between a computer's memory and its processor determines the speed of a computer

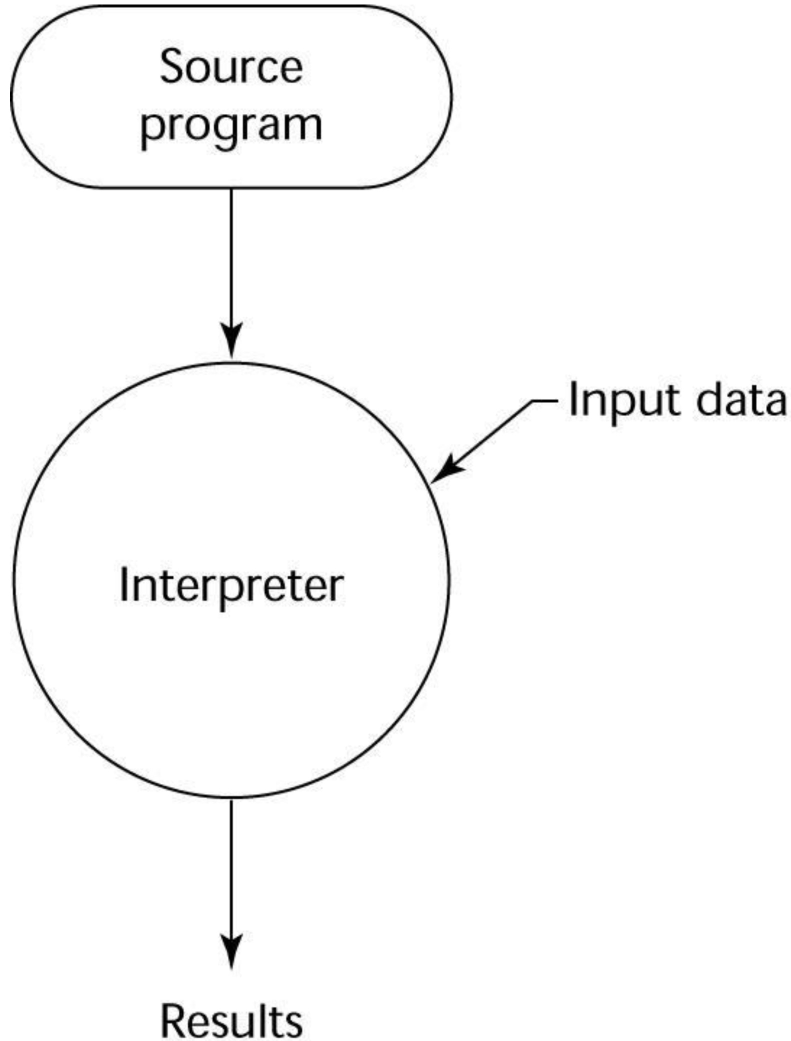
Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*

Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

Pure Interpretation

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

Pure Interpretation Process



Hybrid Implementation Systems

A compromise between compilers and pure interpreters

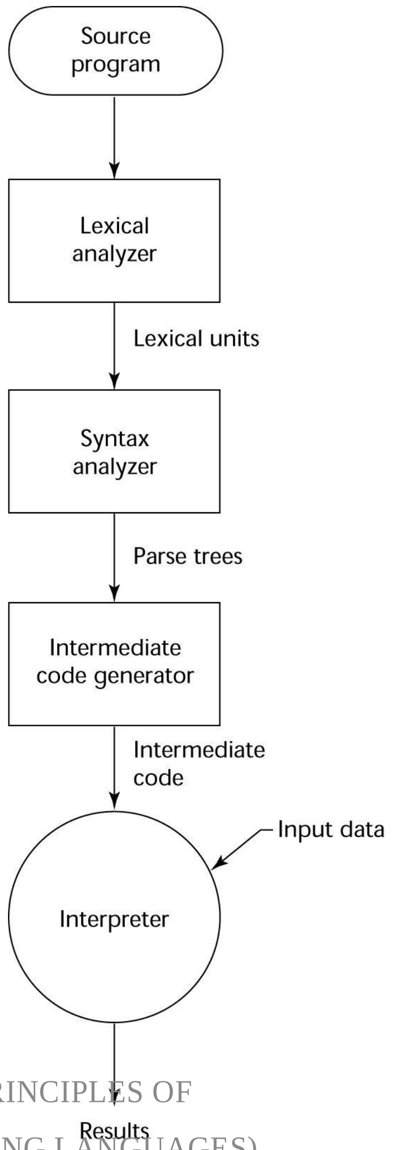
A high-level language program is translated to an intermediate language that allows easy interpretation

Faster than pure interpretation

Examples

- Perl programs are partially compiled to detect errors before interpretation
- Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process



Just-in-Time Implementation Systems

Initially translate programs to an intermediate language

Then compile the intermediate language of the subprograms into machine code when they are called

Machine code version is kept for subsequent calls

JIT systems are widely used for Java programs

.NET languages are implemented with a JIT system

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
  – expands `#include`, `#define`, and similar macros

Programming Environments

A collection of tools used in software development

UNIX

- An older operating system and tool collection
- Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX

Microsoft Visual Studio.NET

- A large, complex visual environment

Used to build Web applications and non-Web applications in any .NET language

NetBeans

- Related to Visual Studio .NET, except for Web applications in Java

Zuse's Plankalkül

Minimal Hardware Programming: Pseudocodes

The IBM 704 and Fortran

Functional Programming: LISP

The First Step Toward Sophistication: ALGOL 60

Computerizing Business Records: COBOL

The Beginnings of Timesharing: BASIC

Everything for Everybody: PL/I

Two Early Dynamic Languages: APL and SNOBOL

The Beginnings of Data Abstraction: SIMULA 67

Orthogonal Design: ALGOL 68

Some Early Descendants of the ALGOLs

Programming Based on Logic: Prolog

History's Largest Design Effort: Ada

Object-Oriented Programming: Smalltalk

Combining Imperative ad Object-Oriented Features: C++

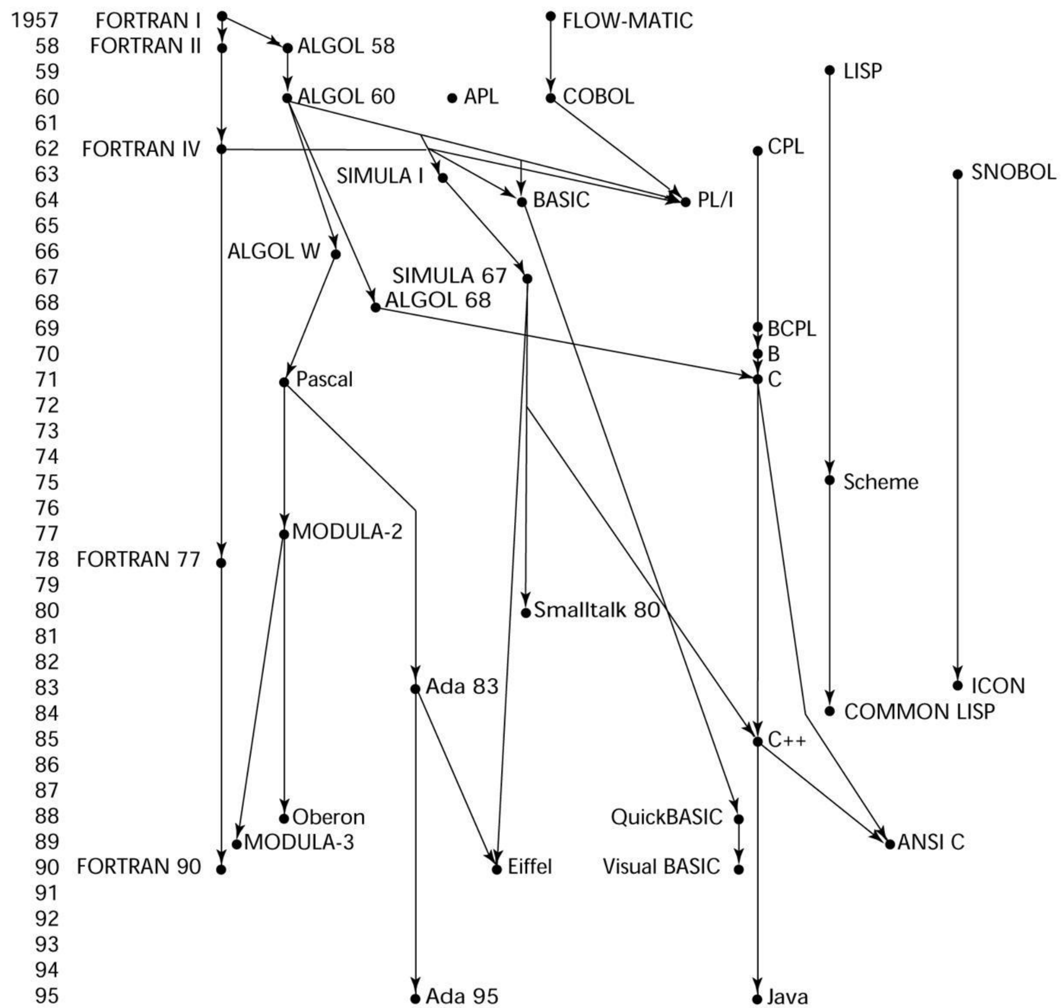
An Imperative-Based Object-Oriented Language: Java

Scripting Languages

A C-Based Language for the New Millennium: C#

Markup/Programming Hybrid Languages

Genealogy of Common Languages



Zuse's Plankalkül

Designed in 1945, but not published until 1972

Never implemented

Advanced data structures

– floating point, arrays, records

Invariants

Plankalkül Syntax

An assignment statement to assign the expression A[4] + 1 to A[5]

```
     | A + 1 => A
  V  | 4       5        (subscripts)
  S  | 1.n     1.n      (data types)
```

Minimal Hardware Programming: Pseudocodes

What was wrong with using machine code?

- Poor readability
- Poor modifiability
- Expression coding was tedious
- Machine deficiencies--no indexing or floating point

Pseudocodes: Short Code

Short Code developed by Mauchly in 1949 for BINAC computers

- Expressions were coded, left to right
- Example of operations:

```
01 – 06 abs value 1n (n+2)nd power
 02  ) 07  +         2n  (n+2)nd root
 03  = 08  pause     4n  if <= n
 04  / 09  (         58  print and tab
```

Pseudocodes: Speedcoding

Speedcoding developed by Backus in 1954 for IBM 701

- Pseudo ops for arithmetic and math functions
- Conditional and unconditional branching
- Auto-increment registers for array access
- Slow!
- Only 700 words left for user program

Pseudocodes: Related Systems

The UNIVAC Compiling System

– Developed by a team led by Grace Hopper
– Pseudocode expanded into machine code

David J. Wheeler (Cambridge University)

– developed a method of using blocks of re-locatable addresses to solve the problem of absolute addressing

IBM 704 and Fortran

Fortran 0: 1954 - not implemented
Fortran I:1957

– Designed for the new IBM 704, which had index registers and floating point hardware

- This led to the idea of compiled programming languages, because there was no place to hide the cost of interpretation (no floating-point software)

– Environment of development

Computers were small and unreliable

Applications were scientific

No programming methodology or tools

Machine efficiency was the most important concern

Design Process of Fortran

Impact of environment on design of Fortran I

- No need for dynamic storage
- Need good array handling and counting loops
- No string handling, decimal arithmetic, or powerful input/output (for business software)

Fortran I Overview

First implemented version of Fortran

- Names could have up to six characters
- Post-test counting loop (**`DO`**)
- Formatted I/O
- User-defined subprograms
- Three-way selection statement (arithmetic **`IF`**)
- No data typing statements

Fortran I Overview (continued)

First implemented version of FORTRAN

- No separate compilation
- Compiler released in April 1957, after 18 worker-years of effort
- Programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of 704
- Code was very fast
- Quickly became widely used

Fortran II

Distributed in 1958

- Independent compilation
- Fixed the bugs

Fortran IV

Evolved during 1960-62

- Explicit type declarations
- Logical selection statement
- Subprogram names could be parameters
- ANSI standard in 1966

Fortran 77

Became the new standard in 1978

- Character string handling
- Logical loop control statement
- **IF-THEN-ELSE** statement

Fortran 90

Most significant changes from Fortran 77

- Modules
- Dynamic arrays
- Pointers
- Recursion
- **`CASE`** statement
- Parameter type checking

Latest versions of Fortran

Fortran 95 – relatively minor additions, plus some deletions

Fortran 2003 - ditto

Fortran Evaluation

Highly optimizing compilers (all versions before 90)

- Types and storage of all variables are fixed before run time

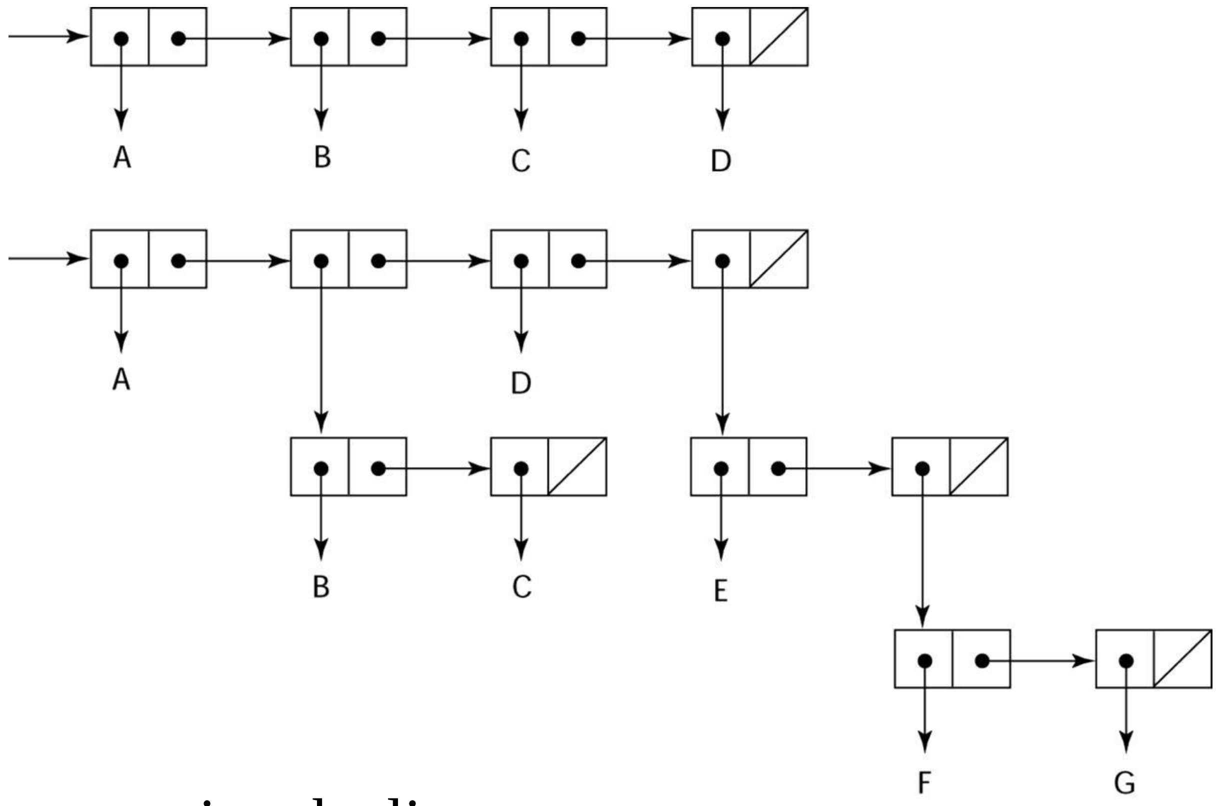
Dramatically changed forever the way computers are used

Characterized as the *lingua franca* of the computing world

Functional Programming: LISP

- LISt Processing language
  - Designed at MIT by McCarthy
- AI research needed a language to
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists
- Syntax is based on *lambda calculus*

Representation of Two LISP Lists



Representing the lists **(A B C D)**
and **(A (B C) D (E (F G)))**

LISP Evaluation

Pioneered functional programming

- No need for variables or assignment
- Control via recursion and conditional expressions

Still the dominant language for AI

COMMON LISP and Scheme are contemporary dialects of LISP

ML, Miranda, and Haskell are related languages

Scheme

Developed at MIT in mid 1970s

Small

Extensive use of static scoping

Functions as first-class entities

Simple syntax (and small size) make it ideal for educational applications

COMMON LISP

An effort to combine features of several dialects of LISP into a single language

Large, complex

The First Step Toward Sophistication: ALGOL 60

- Environment of development
  - FORTRAN had (barely) arrived for IBM 70x
  - Many other languages were being developed, all for specific machines
  - No portable language; all were machine-dependent
  - No universal language for communicating algorithms
- ALGOL 60 was the result of efforts to design a universal language

Early Design Process

ACM and GAMM met for four days for design (May 27 to June 1, 1958)

Goals of the language

- Close to mathematical notation
- Good for describing algorithms
- Must be translatable to machine code

ALGOL 58

Concept of type was formalized

Names could be any length

Arrays could have any number of subscripts

Parameters were separated by mode (in & out)

Subscripts were placed in brackets

Compound statements (**`begin ... end`**)

Semicolon as a statement separator

Assignment operator was :=

`if` had an **`else-if`** clause

No I/O - “would make it machine dependent”

ALGOL 58 Implementation

- Not meant to be implemented, but variations of it were (MAD, JOVIAL)
- Although IBM was initially enthusiastic, all support was dropped by mid 1959

ALGOL 60 Overview

Modified ALGOL 58 at 6-day meeting in Paris

New features

- Block structure (local scope)
- Two parameter passing methods
- Subprogram recursion
- Stack-dynamic arrays

- Still no I/O and no string handling

ALGOL 60 Evaluation

Successes

– It was the standard way to publish algorithms for over 20 years
– All subsequent imperative languages are based on it
– First machine-independent language
– First language whose syntax was formally defined (BNF)

ALGOL 60 Evaluation (continued)

Failure

- Never widely used, especially in U.S.
- Reasons

  Lack of I/O and the character set made programs non-portable

  Too flexible--hard to implement

  Entrenchment of Fortran

  Formal syntax description

  Lack of support from IBM

Computerizing Business Records: COBOL

Environment of development

- UNIVAC was beginning to use FLOW-MATIC
- USAF was beginning to use AIMACO
- IBM was developing COMTRAN

COBOL Historical Background

- Based on FLOW-MATIC
- FLOW-MATIC features
  - Names up to 12 characters, with embedded hyphens
  - English names for arithmetic operators (no arithmetic expressions)
  - Data and code were completely separate
  - The first word in every statement was a verb

COBOL Design Process

- First Design Meeting (Pentagon) - May 1959
- Design goals
  – Must look like simple English
  – Must be easy to use, even if that means it will be less powerful
  – Must broaden the base of computer users
  – Must not be biased by current compiler problems
- Design committee members were all from computer manufacturers and DoD branches
- Design Problems: arithmetic expressions? subscripts? Fights among manufacturers

COBOL Evaluation

Contributions

- First macro facility in a high-level language
- Hierarchical data structures (records)
- Nested selection statements
- Long names (up to 30 characters), with hyphens
- Separate data division

COBOL: DoD Influence

- First language required by DoD
  – would have failed without DoD
- Still the most widely used business applications language

The Beginning of Timesharing: BASIC

- Designed by Kemeny & Kurtz at Dartmouth
- Design Goals:
  - Easy to learn and use for non-science students
  - Must be "pleasant and friendly"
  - Fast turnaround for homework
  - Free and private access
  - User time is more important than computer time
- Current popular dialect: Visual BASIC
- First widely used language with time sharing

2.8 Everything for Everybody: PL/I

- Designed by IBM and SHARE
- Computing situation in 1964 (IBM's point of view)
  - Scientific computing
    - IBM 1620 and 7090 computers
    - FORTRAN
    - SHARE user group
  - Business computing
    - IBM 1401, 7080 computers
    - COBOL
    - GUIDE user group

PL/I: Background

By 1963

- Scientific users began to need more elaborate I/O, like COBOL had; business users began to need floating point and arrays for MIS
- It looked like many shops would begin to need two kinds of computers, languages, and support staff--too costly

The obvious solution

- Build a new computer to do both kinds of applications
- Design a new language to do both kinds of applications

PL/I: Design Process

- Designed in five months by the 3 X 3 Committee
  - Three members from IBM, three members from SHARE
- Initial concept
  - An extension of Fortran IV
- Initially called NPL (New Programming Language)
- Name changed to PL/I in 1965

PL/I: Evaluation

- PL/I contributions
  – First unit-level concurrency
  – First exception handling
  – Switch-selectable recursion
  – First pointer data type
  – First array cross sections
- Concerns
  – Many new features were poorly designed
  – Too large and too complex

Two Early Dynamic Languages: APL and SNOBOL

- Characterized by dynamic typing and dynamic storage allocation
- Variables are untyped
  - A variable acquires a type when it is assigned a value
- Storage is allocated to a variable when it is assigned a value

APL: A Programming Language

Designed as a hardware description language at IBM by Ken Iverson around 1960

- Highly expressive (many operators, for both scalars and arrays of various dimensions)
- Programs are very difficult to read

Still in use; minimal changes

SNOBOL

Designed as a string manipulation language at Bell Labs by Farber, Griswold, and Polensky in 1964

Powerful operators for string pattern matching

Slower than alternative languages (and thus no longer used for writing editors)

Still used for certain text processing tasks

The Beginning of Data Abstraction: SIMULA 67

Designed primarily for system simulation in Norway by Nygaard and Dahl

Based on ALGOL 60 and SIMULA I

Primary Contributions

- Coroutines - a kind of subprogram
- Classes, objects, and inheritance

Orthogonal Design: ALGOL 68

From the continued development of ALGOL 60 but not a superset of that language

Source of several new ideas (even though the language itself never achieved widespread use)

Design is based on the concept of orthogonality

- A few basic concepts, plus a few combining mechanisms

ALGOL 68 Evaluation

Contributions

- User-defined data structures
- Reference types
- Dynamic arrays (called flex arrays)

Comments

- Less usage than ALGOL 60
- Had strong influence on subsequent languages, especially Pascal, C, and Ada

Pascal - 1971

- Developed by Wirth (a former member of the ALGOL 68 committee)
- Designed for teaching structured programming
- Small, simple, nothing really new
- Largest impact was on teaching programming
  - From mid-1970s until the late 1990s, it was the most widely used language for teaching programming

C - 1972

Designed for systems programming (at Bell Labs by Dennis Richie)

Evolved primarily from BCLP, B, but also ALGOL 68

Powerful set of operators, but poor type checking

Initially spread through UNIX

Many areas of application

Programming Based on Logic: Prolog

- Developed, by Comerauer and Roussel (University of Aix-Marseille), with help from Kowalski (University of Edinburgh)
- Based on formal logic
- Non-procedural
- Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries
- Highly inefficient, small application areas

History's Largest Design Effort: Ada

- Huge design effort, involving hundreds of people, much money, and about eight years
  – Strawman requirements (April 1975)
  – Woodman requirements (August 1975)
  – Tinman requirements (1976)
  – Ironman equipments (1977)
  – Steelman requirements (1978)
- Named Ada after Augusta Ada Byron, the first programmer

Ada Evaluation

Contributions

- Packages - support for data abstraction
- Exception handling - elaborate
- Generic program units
- Concurrency - through the tasking model

Comments

- Competitive design
- Included all that was then known about software engineering and language design
- First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed

Ada 95

Ada 95 (began in 1988)

- Support for OOP through type derivation
- Better control mechanisms for shared data
- New concurrency features
- More flexible libraries

Popularity suffered because the DoD no longer requires its use but also because of popularity of C++

Object-Oriented Programming: Smalltalk

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg
- First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic binding)
- Pioneered the graphical user interface design
- Promoted OOP

Combining Imperative and Object-Oriented Programming: C++

- Developed at Bell Labs by Stroustrup in 1980
- Evolved from C and SIMULA 67
- Facilities for object-oriented programming, taken partially from SIMULA 67
- Provides exception handling
- A large and complex language, in part because it supports both procedural and OO programming
- Rapidly grew in popularity, along with OOP
- ANSI standard approved in November 1997
- Microsoft's version (released with .NET in 2002): Managed C++
  - delegates, interfaces, no multiple inheritance

Related OOP Languages

Eiffel (designed by Bertrand Meyer - 1992)

- Not directly derived from any other language
- Smaller and simpler than C++, but still has most of the power
- Lacked popularity of C++ because many C++ enthusiasts were already C programmers

Delphi (Borland)

- Pascal plus features to support OOP
- More elegant and safer than C++

An Imperative-Based Object-Oriented Language: Java

- Developed at Sun in the early 1990s
  - C and C++ were not satisfactory for embedded electronic devices
- Based on C++
  - Significantly simplified (does not include **`struct, union, enum`**, pointer arithmetic, and half of the assignment coercions of C++)
  - Supports *only* OOP
  - Has references, but not pointers
  - Includes support for applets and a form of concurrency

Java Evaluation

Eliminated many unsafe features of C++
Supports concurrency
Libraries for applets, GUIs, database access
Portable: Java Virtual Machine concept, JIT compilers
Widely used for Web programming
Use increased faster than any previous language
Most recent version, 5.0, released in 2004

Scripting Languages for the Web

Perl
- Designed by Larry Wall—first released in 1987
- Variables are statically typed but implicitly declared
- Three distinctive namespaces, denoted by the first character of a variable's name
- Powerful, but somewhat dangerous
- Gained widespread use for CGI programming on the Web
- Also used for a replacement for UNIX system administration language

JavaScript
- Began at Netscape, but later became a joint venture of Netscape and Sun Microsystems
- A client-side HTML-embedded scripting language, often used to create dynamic HTML documents
- Purely interpreted
- Related to Java only through similar syntax

PHP
- PHP: Hypertext Preprocessor, designed by Rasmus Lerdorf
- A server-side HTML-embedded scripting language, often used for form processing and database access through the Web
- Purely interpreted

Scripting Languages for the Web

Python
- An OO interpreted scripting language
- Type checked but dynamically typed
- Used for CGI programming and form processing
- Dynamically typed, but type checked
- Supports lists, tuples, and hashes

Lua
- An OO interpreted scripting language
- Type checked but dynamically typed
- Used for CGI programming and form processing
- Dynamically typed, but type checked
- Supports lists, tuples, and hashes, all with its single data structure, the table
- Easily extendable

Scripting Languages for the Web

Ruby

- Designed in Japan by Yukihiro Matsumoto (a.k.a, "Matz")
- Began as a replacement for Perl and Python
- A pure object-oriented scripting language

    All data are objects

- Most operators are implemented as methods, which can be redefined by user code
- Purely interpreted

C-Based Language for the New Millennium: C#

Part of the .NET development platform (2000)

Based on C++ , Java, and Delphi

Provides a language for component-based software development

All .NET languages use Common Type System (CTS), which provides a common class library

Markup/Programming Hybrid Languages

XSLT

- eXtensible Markup Language (XML): a metamarkup language
- eXtensible Stylesheet Language Transformation (XSTL) transforms XML documents for display
- Programming constructs (e.g., looping)

JSP

- Java Server Pages: a collection of technologies to support dynamic Web documents
- servlet: a Java program that resides on a Web server and is enacted when called by a requested HTML document; a servlet's output is displayed by the browser
- JSTL includes programming constructs in the form of HTML elements

Introduction to 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

Syntax and semantics provide a language's definition

- Users of a language definition
  - Other language designers
  - Implementers
  - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

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

A *language* is a set of sentences

A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)

A *token* is a category of lexemes (e.g., identifier)

Formal Definition of Languages

Recognizers

- A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
- Example: syntax analysis part of a compiler

  Detailed discussion of syntax analysis appears in Chapter 4

Generators

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

BNF and Context-Free Grammars

Context-Free Grammars

- Developed by Noam Chomsky in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context-free languages

Backus-Naur Form (1959)

- Invented by John Backus to describe Algol 58
- BNF is equivalent to context-free grammars

BNF Fundamentals

In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols,* or just *terminals*)

Terminals are lexemes or tokens

A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

Nonterminals are often enclosed in angle brackets

- Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

Grammar: a finite non-empty set of rules

BNF Rules

An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ;
<stmts> <stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> -
<term> <term> → <var> | const
```

An Example Derivation

```
<program> => <stmts> => <stmt>
                    => <var> = <expr>
                    => a = <expr>
                    => a = <term> + <term>
                    => a = <var> + <term>
                    => a = b + <term>
                    => a = b + const
```

Derivations

Every string of symbols in a derivation is a *sentential form*

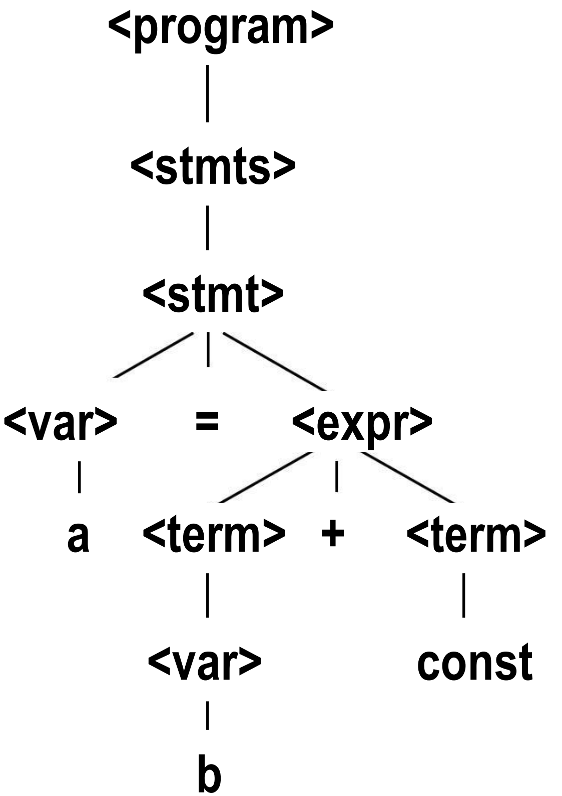
A *sentence* is a sentential form that has only terminal symbols

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

Parse Tree

A hierarchical representation of a derivation

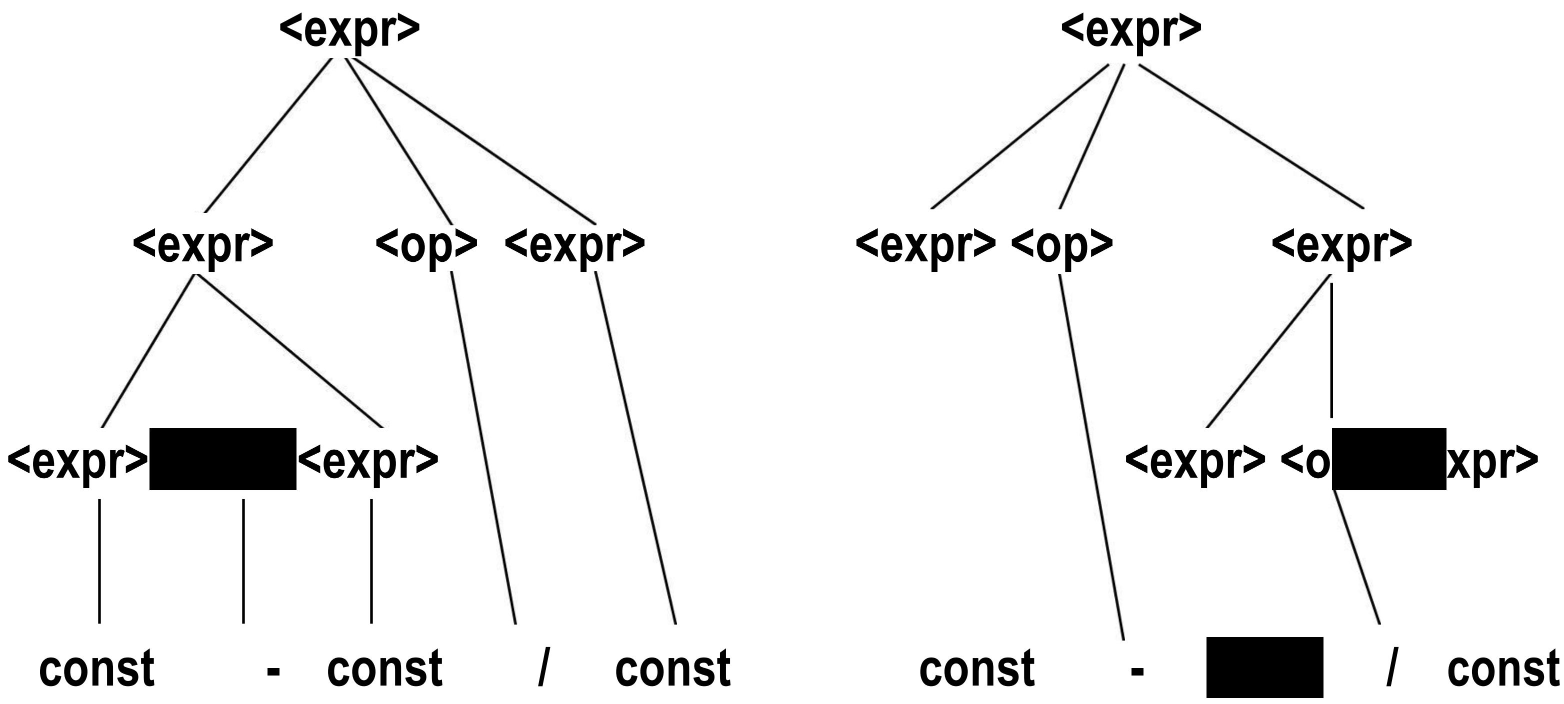


Ambiguity in Grammars

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

An Ambiguous Expression Grammar

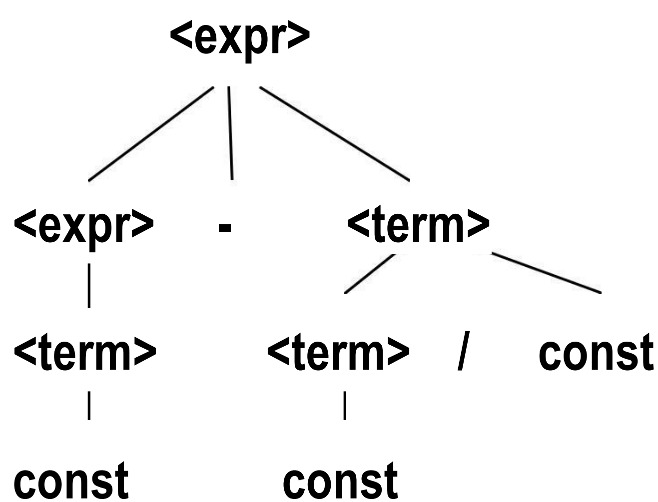
```
<expr> → <expr> <op> <expr> | const
<op> → / | -
```



An Unambiguous Expression Grammar

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

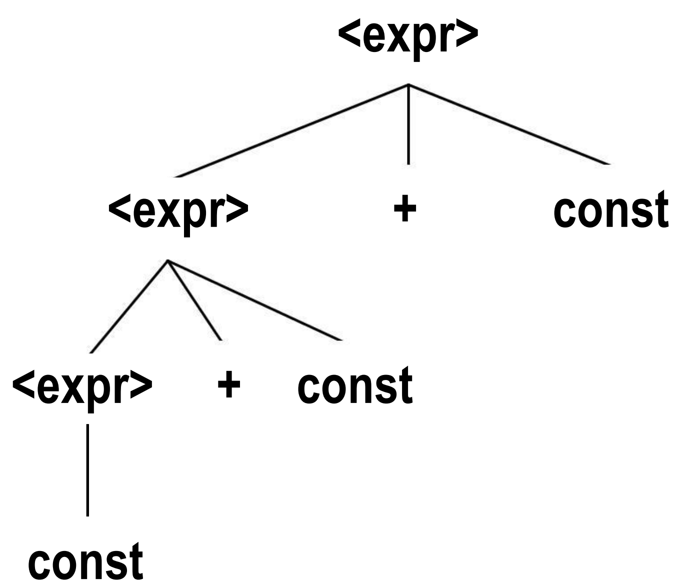
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

Operator associativity can also be indicated by a grammar

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



Extended BNF

Optional parts are placed in brackets [ ]

```
<proc_call> -> ident [(<expr_list>)]
```

Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

BNF and EBNF

BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon instead of `=>`
- Use of `opt` for optional parts
- Use of `oneof` for choices

Static Semantics

- Nothing to do with meaning
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Attribute Grammars

Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

Primary value of AGs:

- Static semantics specification
- Compiler design (static semantics checking)

Attribute Grammars : Definition

Def: An attribute grammar is a context-free grammar 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

Attribute Grammars: Definition

Let $X_0 \to X_1 \ldots X_n$ be a rule

Functions of the form $S(X_0) = f(A(X_1), \ldots , A(X_n))$ define *synthesized attributes*

Functions of the form $I(X_j) = f(A(X_0), \ldots , A(X_n))$, *for* $i \le j \le n$, define *inherited attributes*

Initially, there are *intrinsic attributes* on the leaves

Attribute Grammars: An Example

Syntax

```
<assign> -> <var> = <expr>
<expr> -> <var> + <var> |
<var> <var> A | B | C
```

`actual_type`: synthesized for `<var>` and `<expr>`

`expected_type`: inherited for `<expr>`

Attribute Grammar (continued)

Syntax rule: `<expr> → <var>[1] + <var>[2]`

Semantic rules:

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

Predicate:

```
<var>[1].actual_type == <var>[2].actual_type
<expr>.expected_type == <expr>.actual_type
```

Syntax rule: `<var> → id`

Semantic rule:

`<var>.actual_type ← lookup (<var>.string)`

Attribute Grammars (continued)

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.

Attribute Grammars (continued)

```
<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type
```

Semantics

There is no single widely acceptable notation or formalism for describing semantics

Several needs for a methodology and notation for semantics:

- Programmers need to know what statements mean
- Compiler writers must know exactly what language constructs do
- Correctness proofs would be possible
- Compiler generators would be possible
- Designers could detect ambiguities and inconsistencies

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed

Operational Semantics

A *hardware* pure interpreter would be too expensive

A *software* pure interpreter also has problems

- The detailed characteristics of the particular computer would make actions difficult to understand
- Such a semantic definition would be machine-dependent

Operational Semantics (continued)

A better alternative: A complete computer simulation

The process:

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

Evaluation of operational semantics:

- Good if used informally (language manuals, etc.)
- Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Operational Semantics (continued)

Uses of operational semantics:

Language manuals and textbooks

Teaching programming languages

Two different levels of uses of operational semantics:

Natural operational semantics

Structural operational semantics

Evaluation

Good if used informally (language manuals, etc.)

- Extremely complex if used formally (e.g.,VDL)

Denotational Semantics

Based on recursive function theory

The most abstract semantics description method

Originally developed by Scott and Strachey (1970)

Denotational Semantics - continued

The process of building a denotational specification for a language:

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

Denotational Semantics: program state

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

$$= \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$$

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

$$VARMAP(i_j, s) = v_j$$

Decimal Numbers

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

$$M_{dec}(\text{'0'}) = 0,\ M_{dec}(\text{'1'}) = 1,\ \ldots,\ M_{dec}(\text{'9'}) = 9$$

$$M_{dec}(\text{<dec\_num> '0'}) = 10 * M_{dec}(\text{<dec\_num>})$$

$$M_{dec}(\text{<dec\_num> '1'}) = 10 * M_{dec}(\text{<dec\_num>}) + 1$$

…

$$M_{dec}(\text{<dec\_num> '9'}) = 10 * M_{dec}(\text{<dec\_num>}) + 9$$

Expressions

Map expressions onto Z ∪ {error}

We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

Expressions

```
Me(<expr>, s) Δ=
    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> == '+' 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)
        ...
```

Assignment Statements

Maps state sets to state sets U {`error`}

```
Ma(x := E, s) Δ=
    if Me(E, s) == error
       then error
       else s' =
      {<i1,v1'>,<i2,v2'>,...,<in,vn'>},
              where for j = 1, 2, ..., n,
                 if ij  == x
                   then vj' = Me(E, s)
                   else vj' = VARMAP(ij, s)
```

Logical Pretest Loops

Maps state sets to state sets U {`error`}

```
Ml(while B do L, s) Δ=
    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))
```

Loop Meaning

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

In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

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

Evaluation of Denotational Semantics

Can be used to prove the correctness of programs

Provides a rigorous way to think about programs

Can be an aid to language design

Has been used in compiler generation systems

Because of its complexity, it are of little use to language users

Axiomatic Semantics

Based on formal logic (predicate calculus)

Original purpose: formal program verification

Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)

The logic expressions are called *assertions*

Axiomatic Semantics (continued)

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

An assertion following a statement is a *postcondition*

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

Axiomatic Semantics Form

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 entire program is the desired result

- Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

Axiomatic Semantics: Axioms

An axiom for assignment statements

(x = E): $\{Q_{x\text{->}E}\}$ x = E {Q}

The Rule of Consequence:

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

Axiomatic Semantics: Axioms

An inference rule for sequences of the form S1; S2

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

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

Axiomatic Semantics: Axioms

An inference rule for logical pretest loops

{P} while B do S end {Q}

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

where I is the loop invariant (the inductive hypothesis)

Axiomatic Semantics: Axioms

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   can be difficult to prove

Loop Invariant

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

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

Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotation Semantics vs Operational Semantics

In operational semantics, the state changes are defined by coded algorithms

In denotational semantics, the state changes are defined by rigorous mathematical functions

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, axiomatic, denotational

Development, development environment, and evaluation of a number of important programming languages

Perspective into current issues in language design

- The study of programming languages is valuable for a number of reasons:
  – Increase our capacity to use different constructs
  – Enable us to choose languages more intelligently
  – Makes learning new languages easier
- Most important criteria for evaluating programming languages include:
  – Readability, writability, reliability, cost
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation

UNIT-2

Data Types

Expressions and Statements

CONCEPTS

Introduction
Primitive Data Types
Character String Types
User-Defined Ordinal Types
Array Types
Associative Arrays
Record Types
Union Types
Pointer and Reference Types

CONCEPTS

Introduction

Names

Variables

The concept of binding

Scope

Scope and lifetime

Referencing Environments

Named constants

Introduction

A *data type* defines a collection of data objects and a set of predefined operations on those objects

A *descriptor* is the collection of the attributes of a variable

An *object* represents an instance of a user-defined (abstract data) type

One design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

Almost all programming languages provide a set of *primitive data types*

Primitive data types: Those not defined in terms of other data types

Some primitive data types are merely reflections of the hardware

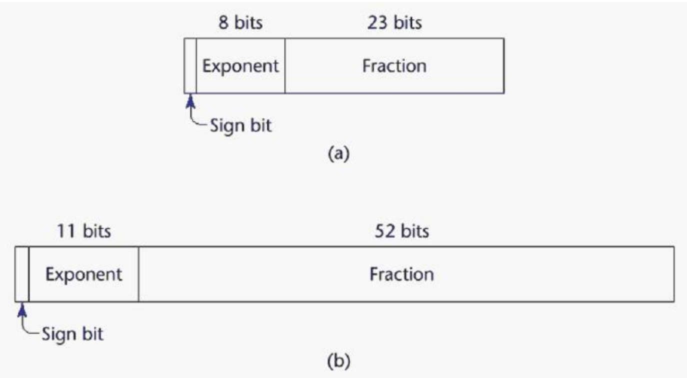
Others require only a little non-hardware support for their implementation

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- Java's signed integer sizes: `byte`, `short`, `int`, `long`

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Primitive Data Types: Complex

Some languages support a complex type, e.g., C99, Fortran, and Python

Each value consists of two floats, the real part and the imaginary part

Literal form (in Python):

`(7 + 3j)`, where `7` is the real part and `3` is the imaginary part

Primitive Data Types: Decimal

For business applications (money)

- Essential to COBOL
- C# offers a decimal data type

Store a fixed number of decimal digits, in coded form (BCD)

Advantage: accuracy

Disadvantages: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes
  – Advantage: readability

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode (UCS-2)
  – Includes characters from most natural languages
  – Originally used in Java
  – C# and JavaScript also support Unicode
- 32-bit Unicode (UCS-4)
  – Supported by Fortran, starting with 2003

Character String Types

- Values are sequences of characters
- Design issues:
  – Is it a primitive type or just a special kind of array?
  – Should the length of strings be static or dynamic?

Character String Types Operations

Typical operations:

– Assignment and copying
– Comparison (=, >, etc.)
– Catenation
– Substring reference
– Pattern matching

Character String Type in Certain Languages

C and C++
- Not primitive
- Use **`char`** arrays and a library of functions that provide operations

SNOBOL4 (a string manipulation language)
- Primitive
- Many operations, including elaborate pattern matching

Fortran and Python
- Primitive type with assignment and several operations

Java
- Primitive via the `String` class

Perl, JavaScript, Ruby, and PHP

Provide built-in pattern matching, using regular expressions

Character String Length Options

- Static: COBOL, Java's `String` class
- *Limited Dynamic Length*: C and C++
  - In these languages, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Character String Type Evaluation

Aid to writability

As a primitive type with static length, they are inexpensive to provide--why not have them?

Dynamic length is nice, but is it worth the expense?

Character String Implementation

Static length: compile-time descriptor

Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)

Dynamic length: need run-time descriptor; allocation/de-allocation is the biggest implementation problem

Compile- and Run-Time Descriptors

Static string
Length
Address

Compile-time descriptor for static strings

Limited dynamic string
Maximum length
Current length
Address

Run-time descriptor for limited dynamic strings

User-Defined Ordinal Types

An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers

Examples of primitive ordinal types in Java

- `integer`
- `char`
- `boolean`

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C# example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Design issues
  – Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
  – Are enumeration values coerced to integer?
  – Any other type coerced to an enumeration type?

Evaluation of Enumerated Type

Aid to readability, e.g., no need to code a color as a number

Aid to reliability, e.g., compiler can check:

- operations (don't allow colors to be added)
- No enumeration variable can be assigned a value outside its defined range
- Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Subrange Types

- An ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type
- Ada's design

```
type Days is (mon, tue, wed, thu, fri, sat,
sun); subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := Day1;
```

Subrange Evaluation

- Aid to readability
  – Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
  – Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementation of User-Defined Ordinal Types

Enumeration types are implemented as integers

Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Array Types

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

Array Design Issues

What types are legal for subscripts?

Are subscripting expressions in element references range checked?

When are subscript ranges bound?

When does allocation take place?

What is the maximum number of subscripts?

Can array objects be initialized?

Are any kind of slices supported?

Array Indexing

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

`array_name (index_value_list) → an element`

Index Syntax

– FORTRAN, PL/I, Ada use parentheses

Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*

– Most other languages use brackets

Arrays Index (Subscript) Types

FORTRAN, C: integer only

Ada: integer or enumeration (includes Boolean and char)

Java: integer types only

Index range checking

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

Subscript Binding and Array Categories

Static: subscript ranges are statically bound and storage allocation is static (before run-time)

– Advantage: efficiency (no dynamic allocation)

Fixed stack-dynamic: subscript ranges are statically bound, but the allocation is done at declaration time

– Advantage: space efficiency

Subscript Binding and Array Categories (continued)

Stack-dynamic: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)

– Advantage: flexibility (the size of an array need not be known until the array is to be used)

Fixed heap-dynamic: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

Subscript Binding and Array Categories (continued)

- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)

Subscript Binding and Array Categories (continued)

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

Array Initialization

Some language allow initialization at the time of storage allocation

- C, C++, Java, C# example

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

- Character strings in C and C++

```
char name [] = “freddie”;
```

- Arrays of strings in C and C++

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

- Java initialization of String objects

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

Heterogeneous Arrays

A *heterogeneous array* is one in which the elements need not be of the same type

Supported by Perl, Python, JavaScript, and Ruby

Array Initialization

C-based languages

- `int list [] = {1, 3, 5, 7}`
- `char *names [] = {“Mike”, “Fred”,“Mary Lou”};`

Ada

- `List : array (1..5) of Integer :=`
  `(1 => 17, 3 => 34, others => 0);`

Python

- List comprehensions

`list = [x ** 2 for x in range(12) if x % 3 == 0]` puts `[0, 9, 36, 81]` in `list`

Arrays Operations

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

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
  – Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)

Slices

A slice is some substructure of an array; nothing more than a referencing mechanism

Slices are only useful in languages that have array operations

Slice Examples

Fortran 95

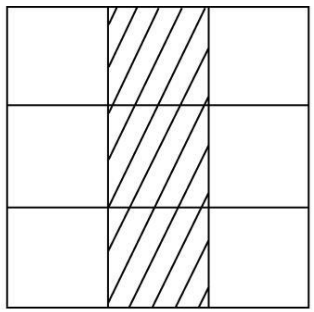
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` is a four element array

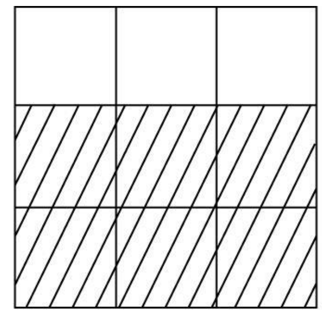
Ruby supports slices with the `slice` method

`list.slice(2, 2)` returns the third and fourth elements of `list`

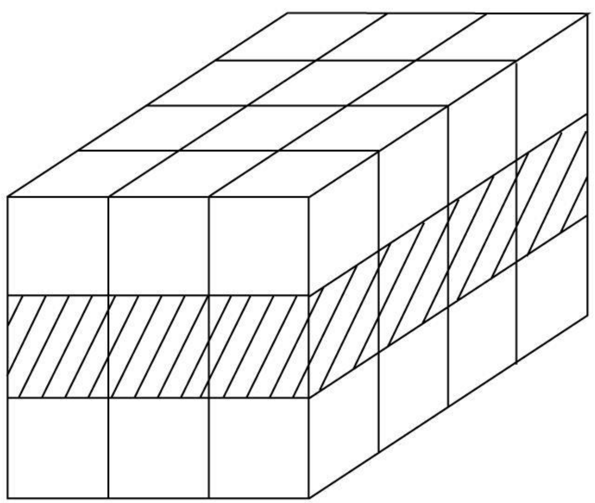
Slices Examples in Fortran 95



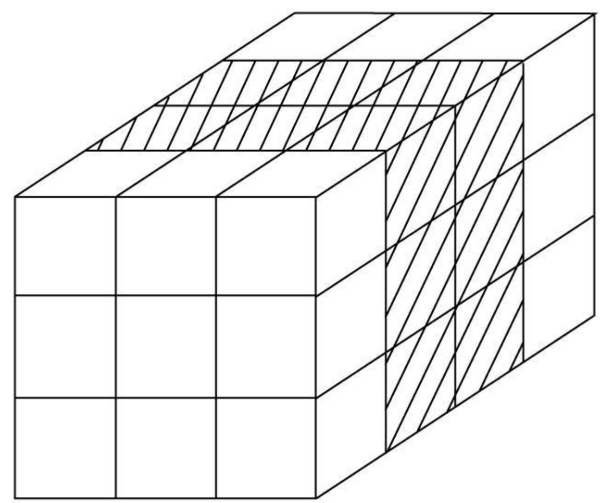
MAT (1:3, 2)



MAT (2:3, 1:3)



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



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

Implementation of Arrays

Access function maps subscript expressions to an address in the array

Access function for single-dimensioned arrays:

address(list[k]) = address (list[lower_bound])

((k-lower_bound) * element_size)

Accessing Multi-dimensioned Arrays

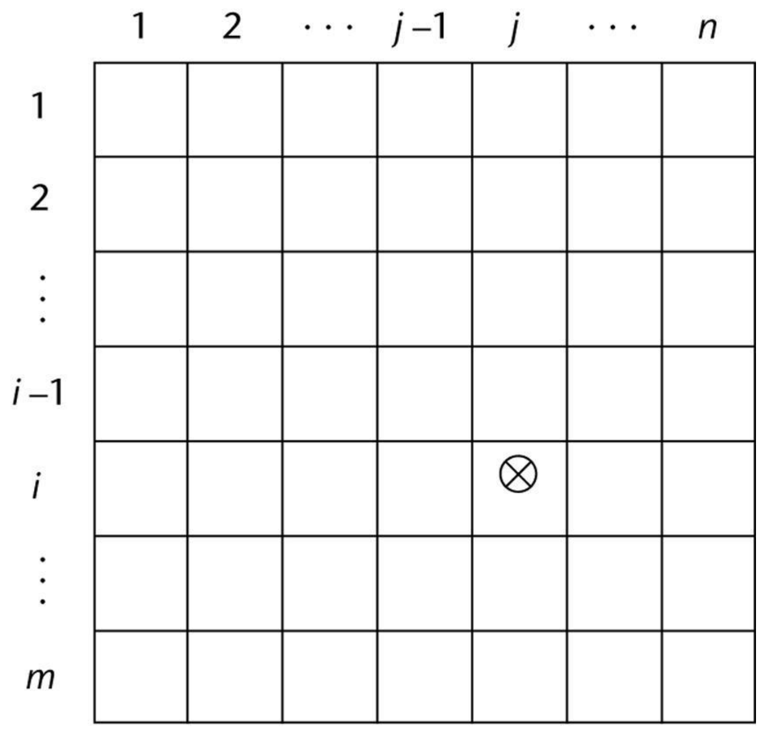
Two common ways:

- Row major order (by rows) – used in most languages
- column major order (by columns) – used in Fortran

Locating an Element in a Multi-dimensioned Array

- General format

  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb)) * element_size



Compile-Time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Multi-dimensional array

Associative Arrays

An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*

- User-defined keys must be stored

Design issues:

What is the form of references to elements?

Is the size static or dynamic?

Built-in type in Perl, Python, Ruby, and Lua

- In Lua, they are supported by tables

Associative Arrays in Perl

Names begin with `%;` literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue"
  => 79, “Wed” => 65, …);
```

Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

– Elements can be removed with `delete`

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

Record Types

A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names

Design issues:

- What is the syntactic form of references to the field?
- Are elliptical references allowed

Definition of Records in COBOL

COBOL uses level numbers to show nested records; others use recursive definition

```
EMP-REC.
02 EMP-NAME.
    05 FIRST PIC X(20).
    05 MID   PIC X(10).
    05 LAST  PIC X(20).
02 HOURLY-RATE PIC 99V99.
```

Definition of Records in Ada

Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
   First: String (1..20);
   Mid: String (1..10);
   Last: String (1..20);
   Hourly_Rate: Float;
end record; Emp_Rec:
Emp_Rec_Type;
```

References to Records

Record field references

1. COBOL

field_name OF record_name_1 OF ... OF record_name_n

2. Others (dot notation)

record_name_1.record_name_2. ... record_name_n.field_name

Fully qualified references must include all record names

Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL

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

Operations on Records

- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  – Copies a field of the source record to the corresponding field in the target record

Evaluation and Comparison to Arrays

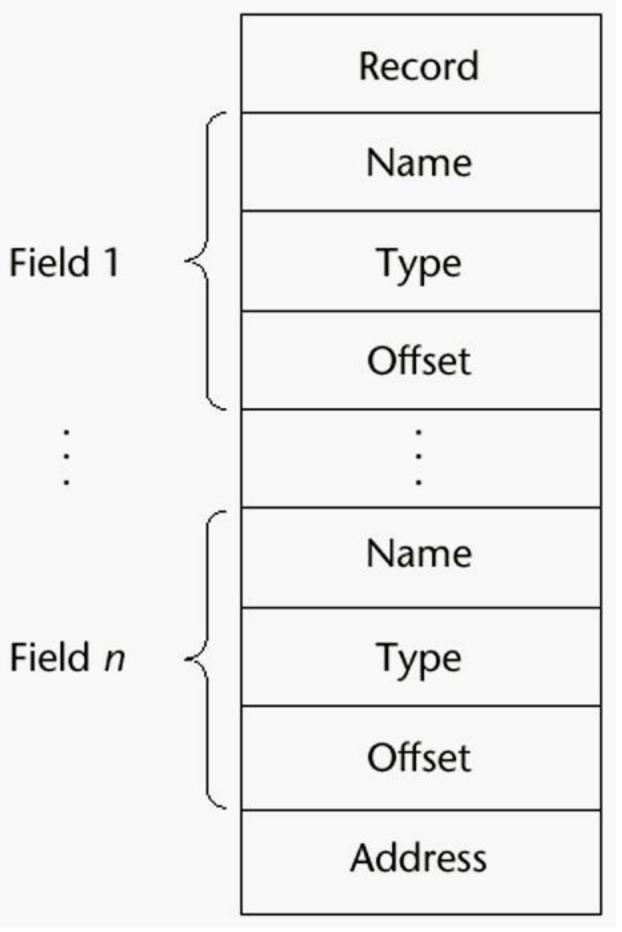
Records are used when collection of data values is heterogeneous

Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)

Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Unions Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issues
  – Should type checking be required?
  – Should unions be embedded in records?

Discriminated vs. Free Unions

Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*

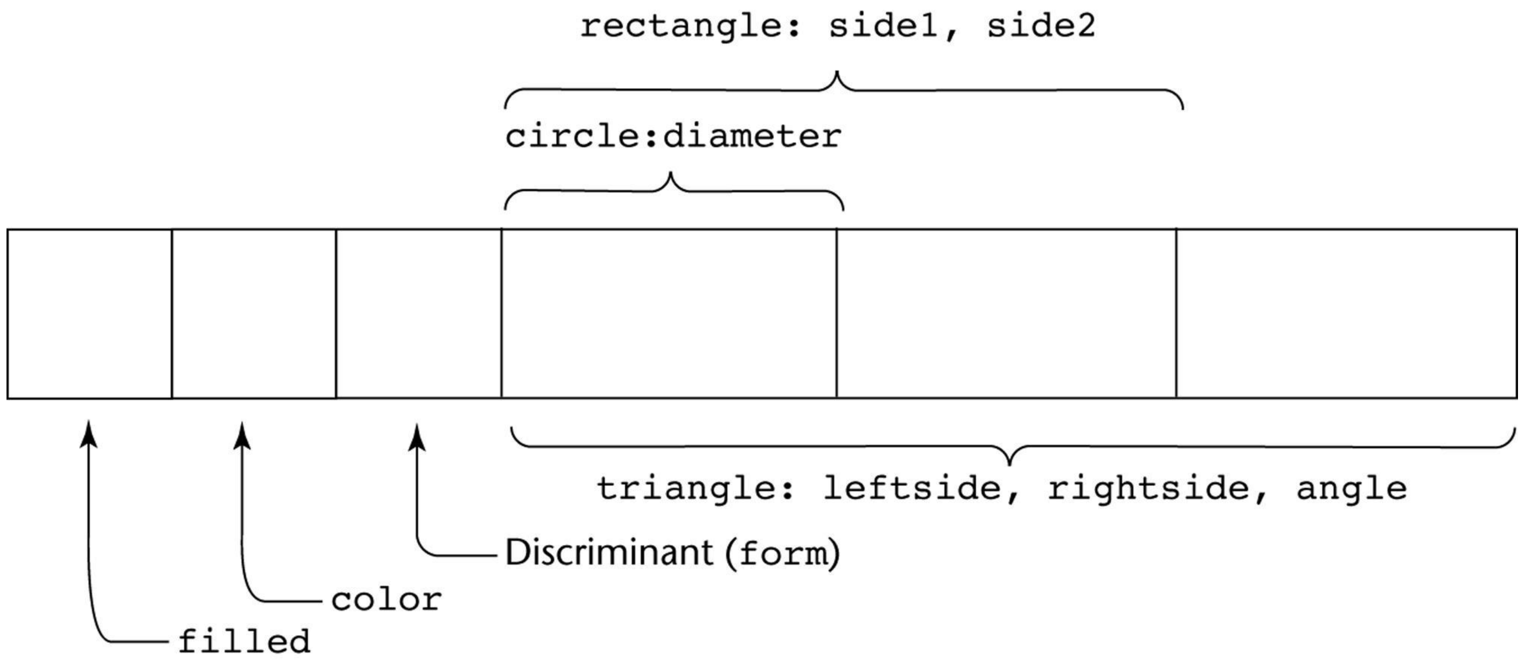
Type checking of unions require that each union include a type indicator called a *discriminant*

– Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
   Filled: Boolean;
   Color: Colors;
   case Form is
        when Circle => Diameter:
        Float; when Triangle =>
              Leftside, Rightside: Integer;
              Angle: Float;
        when Rectangle => Side1, Side2: Integer;
   end case;
end record;
```

Ada Union Type Illustrated



A discriminated union of three shape variables

Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language
- Ada's descriminated unions are safe

Pointer and Reference Types

A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*

Provide the power of indirect addressing

Provide a way to manage dynamic memory

A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

Design Issues of Pointers

What are the scope of and lifetime of a pointer variable?

What is the lifetime of a heap-dynamic variable?

Are pointers restricted as to the type of value to which they can point?

Are pointers used for dynamic storage management, indirect addressing, or both?

Should the language support pointer types, reference types, or both?

Pointer Operations

Two fundamental operations: assignment and dereferencing

Assignment is used to set a pointer variable's value to some useful address

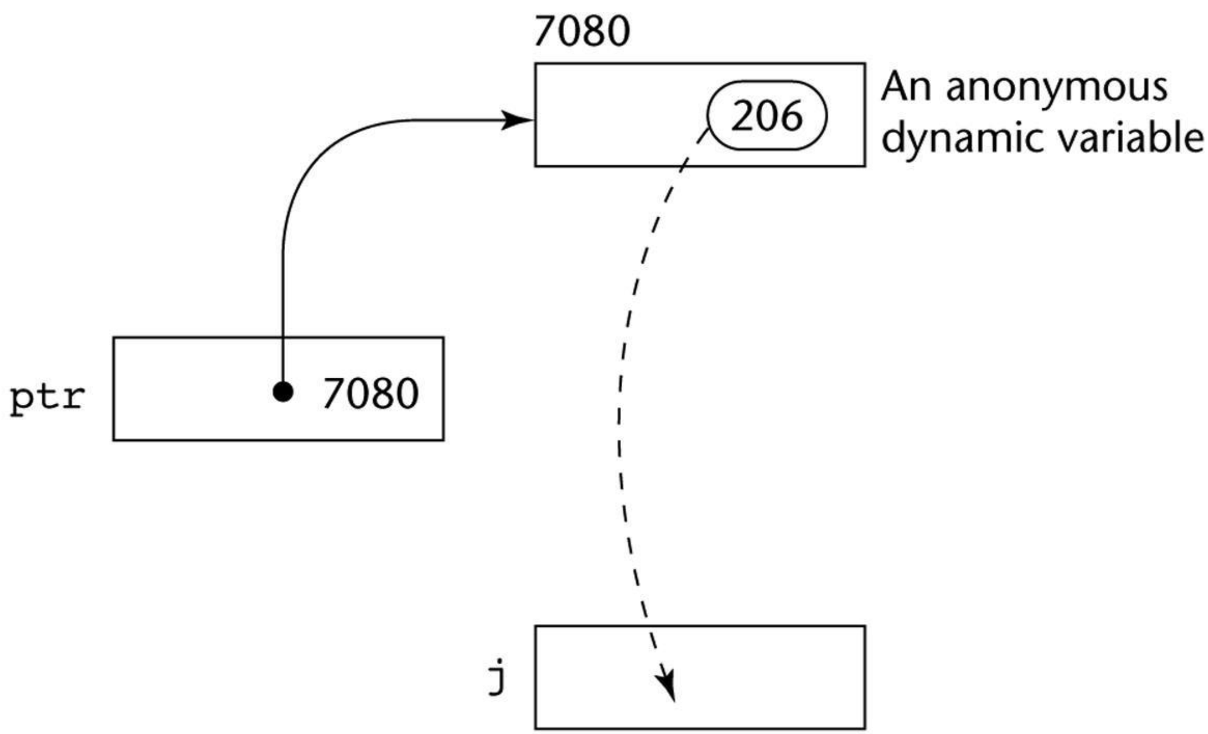
Dereferencing yields the value stored at the location represented by the pointer's value

- Dereferencing can be explicit or implicit
- C++ uses an explicit operation via

  `* j = *ptr`

  sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

Dangling pointers (dangerous)

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

Lost heap-dynamic variable

– An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

Pointer `p1` is set to point to a newly created heap-dynamic variable

Pointer `p1` is later set to point to another newly created heap-dynamic variable

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

Pointers in Ada

Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope

The lost heap-dynamic variable problem is not eliminated by Ada (possible with `UNCHECKED_DEALLOCATION`)

Pointers in C and C++

Extremely flexible but must be used with care

Pointers can point at any variable regardless of when or where it was allocated

Used for dynamic storage management and addressing

Pointer arithmetic is possible

Explicit dereferencing and address-of operators

Domain type need not be fixed (**`void *`**)

`void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

- `*(p+5)` is equivalent to `stuff[5]` and `p[5]`
- `*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

Dangling pointers and dangling objects are problems as is heap management

Pointers are like `goto`'s--they widen the range of cells that can be accessed by a variable

Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Representations of Pointers

Large computers use single values

Intel microprocessors use segment and offset

Dangling Pointer Problem

Tombstone: extra heap cell that is a pointer to the heap-dynamic variable

- The actual pointer variable points only at tombstones
- When heap-dynamic variable de-allocated, tombstone remains but set to nil
- Costly in time and space

. *Locks-and-keys*: Pointer values are represented as (key, address) pairs

- Heap-dynamic variables are represented as variable plus cell for integer lock value
- When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Heap Management

A very complex run-time process

Single-size cells vs. variable-size cells

Two approaches to reclaim garbage

– Reference counters (*eager approach*): reclamation is gradual

– Mark-sweep (*lazy approach*): reclamation occurs when the list of variable space becomes empty

Reference Counter

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

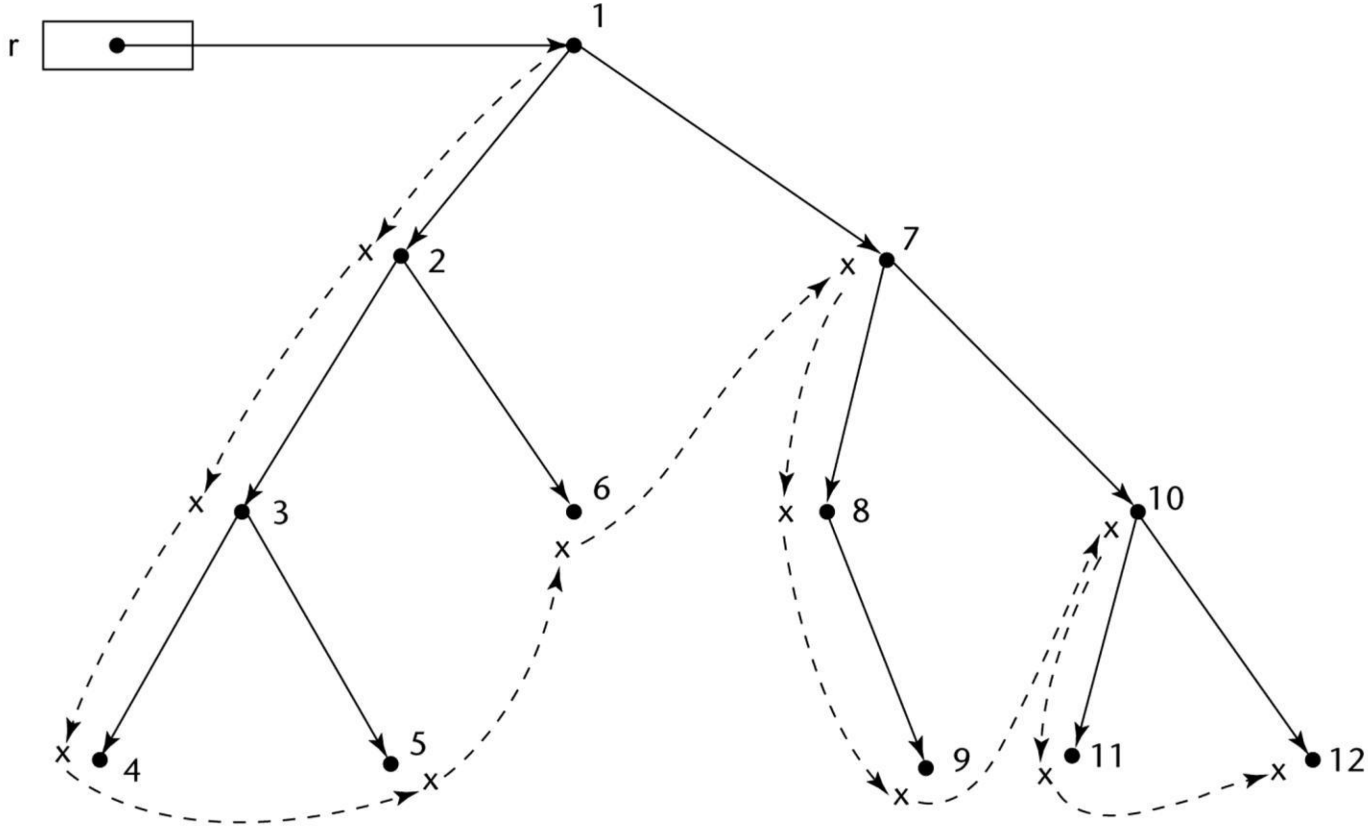
- *Disadvantages*: space required, execution time required, complications for cells connected circularly
- *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided

Mark-Sweep

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

- Every heap cell has an extra bit used by collection algorithm
- All cells initially set to garbage
- All pointers traced into heap, and reachable cells marked as not garbage
- All garbage cells returned to list of available cells
- Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Marking Algorithm



Dashed lines show the order of node_marking

Variable-Size Cells

All the difficulties of single-size cells plus more

Required by most programming languages

If mark-sweep is used, additional problems occur

- The initial setting of the indicators of all cells in the heap is difficult
- The marking process in nontrivial
- Maintaining the list of available space is another source of overhead

Type Checking

Generalize the concept of operands and operators to include subprograms and assignments

Type checking is the activity of ensuring that the operands of an operator are of compatible types

A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type

– This automatic conversion is called a *coercion*.

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

Type Checking (continued)

If all type bindings are static, nearly all type checking can be static

If type bindings are dynamic, type checking must be dynamic

A programming language is *strongly typed* if type errors are always detected

Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

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

  (Java and C# are similar to Ada)

Strong Typing (continued)

Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)

Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Name Type Equivalence

Name type equivalence means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name

Easy to implement but highly restrictive:

- Subranges of integer types are not equivalent with integer types
- Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

Structure type equivalence means that two variables have equivalent types if their types have identical structures

More flexible, but harder to implement

Type Equivalence (continued)

Consider the problem of two structured types:

- Are two record types equivalent if they are structurally the same but use different field names?
- Are two array types equivalent if they are the same except that the subscripts are different? (e.g. [1..10] and [0..9])
- Are two enumeration types equivalent if their components are spelled differently?
- With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Theory and Data Types

Type theory is a broad area of study in mathematics, logic, computer science, and philosophy

Two branches of type theory in computer science:

- Practical – data types in commercial languages
- Abstract – typed lambda calculus

A type system is a set of types and the rules that govern their use in programs

Theory and Data Types (continued)

Formal model of a type system is a set of types and a collection of functions that define the type rules

- Either an attribute grammar or a type map could be used for the functions
- Finite mappings – model arrays and functions
- Cartesian products – model tuples and records
- Set unions – model union types
- Subsets – model subtypes

Introduction

- Imperative languages are abstractions of von Neumann architecture
  – Memory
  – Processor
- Variables characterized by attributes
  – To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

Design issues for names:

– Are names case sensitive?

– Are special words reserved words or keywords?

Names (continued)

Length

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

FORTRAN 95: maximum of 31

C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31

C#, Ada, and Java: no limit, and all are significant

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

Names (continued)

Special characters

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

Names (continued)

Case sensitivity

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

Names (continued)

Special words

- An aid to readability; used to delimit or separate statement clauses

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

    - `Real VarName` (`Real` *is a data type followed with a name, therefore* `Real` *is a keyword)*
    - `Real = 3.4` (`Real` *is a variable)*

- A *reserved word* is a special word that cannot be used as a user-defined name
- Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables

A variable is an abstraction of a memory cell

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Lifetime
- Scope

Variables Attributes

Name - not all variables have them

Address - the memory address with which it is associated

- A variable may have different addresses at different times during execution
- A variable may have different addresses at different places in a program
- If two variable names can be used to access the same memory location, they are called aliases
- Aliases are created via pointers, reference variables, C and C++ unions
- Aliases are harmful to readability (program readers must remember all of them)

Variables Attributes (continued)

Type - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision

Value - the contents of the location with which the variable is associated

The l-value of a variable is its address

The r-value of a variable is its value

Abstract memory cell - the physical cell or collection of cells associated with a variable

The Concept of Binding

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

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

Possible Binding Times

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

Static and Dynamic Binding

A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.

A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Type Binding

How is a type specified?

When does the binding take place?

If static, the type may be specified by either an explicit or an implicit declaration

Explicit/Implicit Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables
- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
- FORTRAN, BASIC, and Perl provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability
  - Disadvantage: reliability (less trouble with Perl)

Dynamic Type Binding

Dynamic Type Binding (JavaScript and PHP)

Specified through an assignment statement e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

- Advantage: flexibility (generic program units)
- Disadvantages:
  - High cost (dynamic type checking and interpretation)
  - Type error detection by the compiler is difficult

Variable Attributes (continued)

- Type Inferencing (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined (by the compiler) from the context of the reference
- Storage Bindings & Lifetime
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool
- The lifetime of a variable is the time during which it is bound to a particular memory cell

Categories of Variables by Lifetimes

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

Categories of Variables by Lifetimes

Stack-dynamic--Storage bindings are created for variables when their declaration statements are *elaborated*.

(A declaration is elaborated when the executable code associated with it is executed)

If scalar, all attributes except address are statically bound

- local variables in C subprograms and Java methods

Advantage: allows recursion; conserves storage

Disadvantages:

- Overhead of allocation and deallocation
- Subprograms cannot be history sensitive
- Inefficient references (indirect addressing)

Categories of Variables by Lifetimes

Explicit heap-dynamic -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution

Referenced only through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java

Advantage: provides for dynamic storage management

Disadvantage: inefficient and unreliable

Categories of Variables by Lifetimes

- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements
  – all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Advantage: flexibility (generic code)
- Disadvantages:
  – Inefficient, because all attributes are dynamic
  – Loss of error detection

Variable Attributes: Scope

The *scope* of a variable is the range of statements over which it is visible

The *nonlocal variables* of a program unit are those that are visible but not declared there

The scope rules of a language determine how references to names are associated with variables

Static Scope

Based on program text

To connect a name reference to a variable, you (or the compiler) must find the declaration

Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name

Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*

Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Fortran 2003, and PHP)

Scope (continued)

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

Blocks

– A method of creating static scopes inside program units--from ALGOL 60

– Example in C:

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

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

Declaration Order

C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear

- In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
- In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
  - However, a variable still must be declared before it can be used

Declaration Order (continued)

- In C++, Java, and C#, variables can be declared in `for` statements
  - The scope of such variables is restricted to the `for` construct

Global Scope

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

Global Scope (continued)

PHP

- Programs are embedded in XHTML markup documents, in any number of fragments, some statements and some function definitions
- The scope of a variable (implicitly) declared in a function is local to the function
- The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions

  Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`

Global Scope (continued)

Python

– A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

Evaluation of Static Scoping

Works well in many situations

Problems:

– In most cases, too much access is possible
– As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

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

Scope Example

```
Big
    declaration of X
      Sub1
        declaration of X -
        ...
        call Sub2
        ...

      Sub2
        ...
        - reference to X -
        ...

    ...
    call Sub1
    ...
```

Big calls Sub1
Sub1 calls Sub2
Sub2 uses X

Scope Example

- Static scoping
  - Reference to X is to Big's X
- Dynamic scoping
  - Reference to X is to Sub1's X
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*

    While a subprogram is executing, its variables are visible to all subprograms it calls

    Impossible to statically type check

    Poor readability- it is not possible to statically determine the type of a variable

Scope and Lifetime

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

Consider a `static` variable in a C or C++ function

Referencing Environments

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

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

A subprogram is active if its execution has begun but has not yet terminated

In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Named Constants

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

Advantages: readability and modifiability

Used to parameterize programs

The binding of values to named constants can be either static (called *manifest constants*) or dynamic

Languages:

- FORTRAN 95: constant-valued expressions
- Ada, C++, and Java: expressions of any kind
- C# has two kinds, `readonly` and `const`

  the values of `const` named constants are bound at compile time

  The values of `readonly` named constants are dynamically bound

Summary

Case sensitivity and the relationship of names to special words represent design issues of names

Variables are characterized by the sextuples: name, address, value, type, lifetime, scope

Binding is the association of attributes with program entities

Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic

Strong typing means detecting all type errors

Introduction

Arithmetic Expressions

Overloaded Operators

Type Conversions

Relational and Boolean Expressions

Short-Circuit Evaluation

Assignment Statements

Mixed-Mode Assignment

Introduction

Expressions are the fundamental means of specifying computations in a programming language

To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation

Essence of imperative languages is dominant role of assignment statements

Arithmetic Expressions

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

Arithmetic Expressions: Design Issues

Design issues for arithmetic expressions

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

Arithmetic Expressions: Operators

A unary operator has one operand

A binary operator has two operands

A ternary operator has three operands

Arithmetic Expressions: Operator Precedence Rules

The *operator precedence rules* for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated

Typical precedence levels

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

Arithmetic Expressions: Operator Associativity Rule

The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated

Typical associativity rules

- Left to right, except **, which is right to left
- Sometimes unary operators associate right to left (e.g., in FORTRAN)

APL is different; all operators have equal precedence and all operators associate right to left

Precedence and associativity rules can be overriden with parentheses

Ruby Expressions

All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods

One result of this is that these operators can all be overriden by application programs

Arithmetic Expressions: Conditional Expressions

Conditional Expressions

- C-based languages (e.g., C, C++)
- An example:

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

- Evaluates as if written like

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

Arithmetic Expressions: Operand Evaluation Order

Operand evaluation order

- Variables: fetch the value from memory
- Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
- Parenthesized expressions: evaluate all operands and operators first
- The most interesting case is when an operand is a function call

Arithmetic Expressions: Potentials for Side Effects

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

Problem with functional side effects:

– When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(&a);
```

Functional Side Effects

Two possible solutions to the problem

- Write the language definition to disallow functional side effects
  - No two-way parameters in functions
  - No non-local references in functions
  - **Advantage:** it works!
  - **Disadvantage:** inflexibility of one-way parameters and lack of non-local references
- Write the language definition to demand that operand evaluation order be fixed
  - **Disadvantage**: limits some compiler optimizations
  - Java requires that operands appear to be evaluated in left-to-right order

Overloaded Operators

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

Overloaded Operators (continued)

C++ and C# allow user-defined overloaded operators

Potential problems:

- Users can define nonsense operations
- Readability may suffer, even when the operators make sense

Type Conversions

A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type e.g., `float` to `int`

A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., `int` to `float`

Type Conversions: Mixed Mode

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

A *coercion* is an implicit type conversion

Disadvantage of coercions:

- They decrease in the type error detection ability of the compiler

In most languages, all numeric types are coerced in expressions, using widening conversions

In Ada, there are virtually no coercions in expressions

Explicit Type Conversions

Called *casting* in C-based languages

Examples

- C: `(int)angle`
- Ada: `Float (Sum)`

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

Type Conversions: Errors in Expressions

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

Relational and Boolean Expressions

Relational Expressions

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

JavaScript and PHP have two additional relational operator, `===` and `!==`

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

Relational and Boolean Expressions

Boolean Expressions

– Operands are Boolean and the result is Boolean

– Example operators

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

Relational and Boolean Expressions: No Boolean Type in C

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

Short Circuit Evaluation

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

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

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

Short Circuit Evaluation (continued)

C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (**`&&`** and **`||`**), but also provide bitwise Boolean operators that are not short circuit (**`&`** and **`|`**)

Ada: programmer can specify either (short-circuit is specified with **`and then`** and **`or else`**)

Short-circuit evaluation exposes the potential problem of side effects in expressions
e.g. **`(a > b) || (b++ / 3)`**

Assignment Statements

The general syntax

`<target_var> <assign_operator> <expression>`

The assignment operator

FORTRAN, BASIC, the C-based languages $=$

$:=$ ALGOLs, Pascal, Ada

= can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use $==$ as the relational operator)

Assignment Statements: Conditional Targets

Conditional targets (Perl)

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

Which is equivalent to

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

Assignment Statements: Compound Operators

A shorthand method of specifying a commonly needed form of assignment

Introduced in ALGOL; adopted by C

Example

```
a = a + b
```

is written as

```
a += b
```

Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
- Examples
  - `sum = ++count` (`count` incremented, added to `sum`)
  - `sum = count++` (`count` incremented, added to `sum`)
  - `count++` (`count` incremented)
  - `-count++` (`count` incremented then negated)

Assignment as an Expression

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

An example:

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

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

List Assignments

Perl and Ruby support list assignments

e.g.,

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

Mixed-Mode Assignment

Assignment statements can also be mixed-mode

In Fortran, C, and C++, any numeric type value can be assigned to any numeric type variable

In Java, only widening assignment coercions are done

In Ada, there is no assignment coercion

Summary

Expressions

Operator precedence and associativity

Operator overloading

Mixed-type expressions

Various forms of assignment

Introduction

Selection Statements

Iterative Statements

Unconditional Branching

Guarded Commands

Conclusions

Levels of Control Flow

- Within expressions (Chapter 7)
- Among program units (Chapter 9)
- Among program statements (this chapter)

Control Statements: Evolution

FORTRAN I control statements were based directly on IBM 704 hardware

Much research and argument in the 1960s about the issue

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

Control Structure

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

Design question

– Should a control structure have multiple entries?

Selection Statements

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

Two general categories:

- Two-way selectors
- Multiple-way selectors

Two-Way Selection Statements

General form:

```
if control_expression
  then clause
  else clause
```

Design Issues:

- What is the form and type of the control expression?
- How are the **`then`** and **`else`** clauses specified?
- How should the meaning of nested selectors be specified?

The Control Expression

If the then reserved word or some other syntactic marker is not used to introduce the then clause, the control expression is placed in parentheses

In C89, C99, Python, and C++, the control expression can be arithmetic

In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean

Clause Form

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

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

Nesting Selectors

- Java example

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

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

Nesting Selectors (continued)

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

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

The above solution is used in C, C++, and C#

Perl requires that all then and else clauses to be compound

Nesting Selectors (continued)

Statement sequences as clauses: Ruby

```
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Nesting Selectors (continued)

Python

```
if sum == 0 :
    if count == 0 :
        result = 0
    else :
        result = 1
```

Multiple-Way Selection Statements

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

Design Issues:

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

Multiple-Way Selection: Examples

C, C++, and Java

```
switch (expression) {
     case const_expr_1: stmt_1;
     …
     case const_expr_n: stmt_n;
     [default: stmt_n+1]
}
```

Multiple-Way Selection: Examples

Design choices for C's **`switch`** statement

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

Multiple-Way Selection: Examples

C#

- Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment

- Each selectable segment must end with an unconditional branch (`goto` or `break`)

- Also, in C# the control expression and the case constants can be strings

Multiple-Way Selection: Examples

Ada

```
case expression is
    when choice list => stmt_sequence;
    …
    when choice list => stmt_sequence;
    when others => stmt_sequence;]
end case;
```

More reliable than C's `switch` (once a stmt_sequence execution is completed, control is passed to the first statement after the `case` statement

Multiple-Way Selection: Examples

Ada design choices:

Expression can be any ordinal type

Segments can be single or compound

Only one segment can be executed per execution of the construct

Unrepresented values are not allowed

Constant List Forms:

A list of constants

Can include:

- Subranges
- Boolean OR operators (|)

Multiple-Way Selection: Examples

Ruby has two forms of case statements

1. One form uses when conditions

```
leap = case
      when year % 400 == 0 then true
      when  year  %  100  ==  0  then
      false else year % 4 == 0
      end
```

2. The other uses a case value and when values

```
case in_val
when -1 then neg_count++
when 0 then zero_count++
when 1 then pos_count++
else puts "Error - in_val is out of range"
end
```

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
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

Multiple-Way Selection Using `if`

- The Python example can be written as a Ruby `case`

```
case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```

Iterative Statements

The repeated execution of a statement or compound statement is accomplished either by iteration or recursion

General design issues for iteration control statements:

- How is iteration controlled?
- Where is the control mechanism in the loop?

Counter-Controlled Loops

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

Design Issues:

- What are the type and scope of the loop variable?
- Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?

Iterative Statements: Examples

- FORTRAN 95 syntax

```
DO label var = start, finish [, stepsize]
```

- Stepsize can be any value but zero
- Parameters can be expressions
- Design choices:
  - Loop variable must be **`INTEGER`**
  - 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

Iterative Statements: Examples

FORTRAN 95 : a second form:

```
[name:] Do variable = initial, terminal [,stepsize]
                 …
End Do [name]
```

Cannot branch into either of Fortran's `Do` statements

Iterative Statements: Examples

Ada

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

Design choices:

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

Loop variable does not exist outside the loop

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

The discrete range is evaluated just once

Cannot branch into the loop body

Iterative Statements: Examples

C-based languages

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

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

- The value of a multiple-statement expression is the value of the last statement in the expression
- If the second expression is absent, it is an infinite loop

Design choices:

There is no explicit loop variable

Everything can be changed in the loop

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

Iterative Statements: Examples

C++ differs from C in two ways:

The control expression can also be Boolean

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

Java and C#

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

Iterative Statements: Examples

Python

```
for loop_variable in object:
  loop body
[else:
  else clause]
```

- The object is often a range, which is either a list of values in brackets ([2, 4, 6]), or a call to the range function (range(5), which returns 0, 1, 2, 3, 4
- The loop variable takes on the values specified in the given range, one for each iteration
- The else clause, which is optional, is executed if the loop terminates normally

Iterative Statements: Logically-Controlled Loops

Repetition control is based on a Boolean expression

Design issues:

- Pretest or posttest?
- Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

Iterative Statements: Logically-Controlled Loops: Examples

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

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

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

Iterative Statements: Logically-Controlled Loops: Examples

Ada has a pretest version, but no posttest

FORTRAN 95 has neither

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

Iterative Statements: User-Located Loop Control Mechanisms

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., `break`)
- Design issues for nested loops
  - Should the conditional be part of the exit?
  - Should control be transferable out of more than one loop?

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

- C , C++, Python, Ruby, and C# have unconditional unlabeled exits (**`break`**)
- Java and Perl have unconditional labeled exits (**`break`** in Java, **`last`** in Perl)
- C, C++, and Python have an unlabeled control statement, **`continue`**, that skips the remainder of the current iteration, but does not exit the loop
- Java and Perl have labeled versions of **`continue`**

Iterative Statements: Iteration Based on Data Structures

- Number of elements of in a data structure control loop iteration
- Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate
- C's **`for`** can be used to build a user-defined iterator:

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

Iterative Statements: Iteration Based on Data Structures (continued)

PHP

`current` points at one element of the array

`next` moves `current` to the next element

`reset` moves `current` to the first element

Java

For any collection that implements the `Iterator` interface

`next` moves the pointer into the collection

`hasNext` is a predicate

`remove` deletes an element

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

Iterative Statements: Iteration Based on Data Structures (continued)

Java 5.0 (uses `for`, although it is called `foreach`)

- For arrays and any other class that implements `Iterable` interface, e.g., `ArrayList`

```
for (String myElement : myList) { … }
```

C#'s **`foreach`** statement iterates on the elements of arrays and other collections:

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

The notation `{0}` indicates the position in the string to be displayed

Iterative Statements: Iteration Based on Data Structures (continued)

Lua

- Lua has two forms of its iterative statement, one like Fortran's `Do`, and a more general form:

```
for variable_1 [, variable_2] in iterator (table) do
   …
 end
```

- The most commonly used iterators are `pairs` and `ipairs`

Unconditional Branching

- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960's and 1970's
- Major concern: Readability
- Some languages do not support `goto` statement (e.g., Java)
- C# offers `goto` statement (can be used in `switch` statements)
- Loop exit statements are restricted and somewhat camouflaged `goto`'s

Guarded Commands

Designed by Dijkstra

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

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

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

Selection Guarded Command

Form

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

Semantics: when construct is reached,

- Evaluate all Boolean expressions
- If more than one are true, choose one non-deterministically
- If none are true, it is a runtime error

Loop Guarded Command

Form

```
do <Boolean> -> <statement>
[] <Boolean> -> <statement>
...
[] <Boolean> -> <statement>
od
```

Semantics: for each iteration

- Evaluate all Boolean expressions
- If more than one are true, choose one non-deterministically; then start loop again
- If none are true, exit loop

Guarded Commands: Rationale

Connection between control statements and program verification is intimate

Verification is impossible with `goto` statements

Verification is possible with only selection and logical pretest loops

Verification is relatively simple with only guarded commands

Summary

The data types of a language are a large part of what determines that language's style and usefulness

The primitive data types of most imperative languages include numeric, character, and Boolean types

The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs

Arrays and records are included in most languages

Pointers are used for addressing flexibility and to control dynamic storage management

Conclusion

Variety of statement-level structures

Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability

Functional and logic programming languages are quite different control structures

Unit-3

Subprograms and Blocks

CONCEPTS

Introduction

Fundamentals of Subprograms

Design Issues for Subprograms

Local Referencing Environments

Parameter-Passing Methods

Parameters That Are Subprograms

Overloaded Subprograms

Generic Subprograms

Design Issues for Functions

User-Defined Overloaded Operators

Coroutines

The General Semantics of Calls and Returns

Implementing "Simple" Subprograms

Implementing Subprograms with Stack-Dynamic Local Variables

Nested Subprograms

Blocks

Implementing Dynamic Scoping

Introduction

Two fundamental abstraction facilities

– Process abstraction

Emphasized from early days

– Data abstraction

Emphasized in the1980s

Fundamentals of Subprograms

Each subprogram has a single entry point

The calling program is suspended during execution of the called subprogram

Control always returns to the caller when the called subprogram's execution terminates

Basic Definitions

A *subprogram definition* describes the interface to and the actions of the subprogram abstraction

In Python, function definitions are executable; in all other languages, they are non-executable

A *subprogram call* is an explicit request that the subprogram be executed

A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters

The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters

The *protocol* is a subprogram's parameter profile and, if it is a function, its return type

Basic Definitions (continued)

Function declarations in C and C++ are often called *prototypes*

A *subprogram declaration* provides the protocol, but not the body, of the subprogram

A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram

An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

Positional

- The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
- Safe and effective

Keyword

- The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
- *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
- *Disadvantage*: User must know the formal parameter's names

Formal Parameter Default Values

In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)

- In C++, default parameters must appear last because parameters are positionally associated

Variable numbers of parameters

- C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**
- In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
- In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
- In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods

Ruby Blocks

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

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

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

Procedures and Functions

There are two categories of subprograms

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

Design Issues for Subprograms

Are local variables static or dynamic?

Can subprogram definitions appear in other subprogram definitions?

What parameter passing methods are provided?

Are parameter types checked?

If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?

Can subprograms be overloaded?

Can subprogram be generic?

Local Referencing Environments

Local variables can be stack-dynamic

Advantages

Support for recursion

Storage for locals is shared among some subprograms

– Disadvantages

Allocation/de-allocation, initialization time

Indirect addressing

Subprograms cannot be history sensitive

Local variables can be static

– Advantages and disadvantages are the opposite of those for stack-dynamic local variables

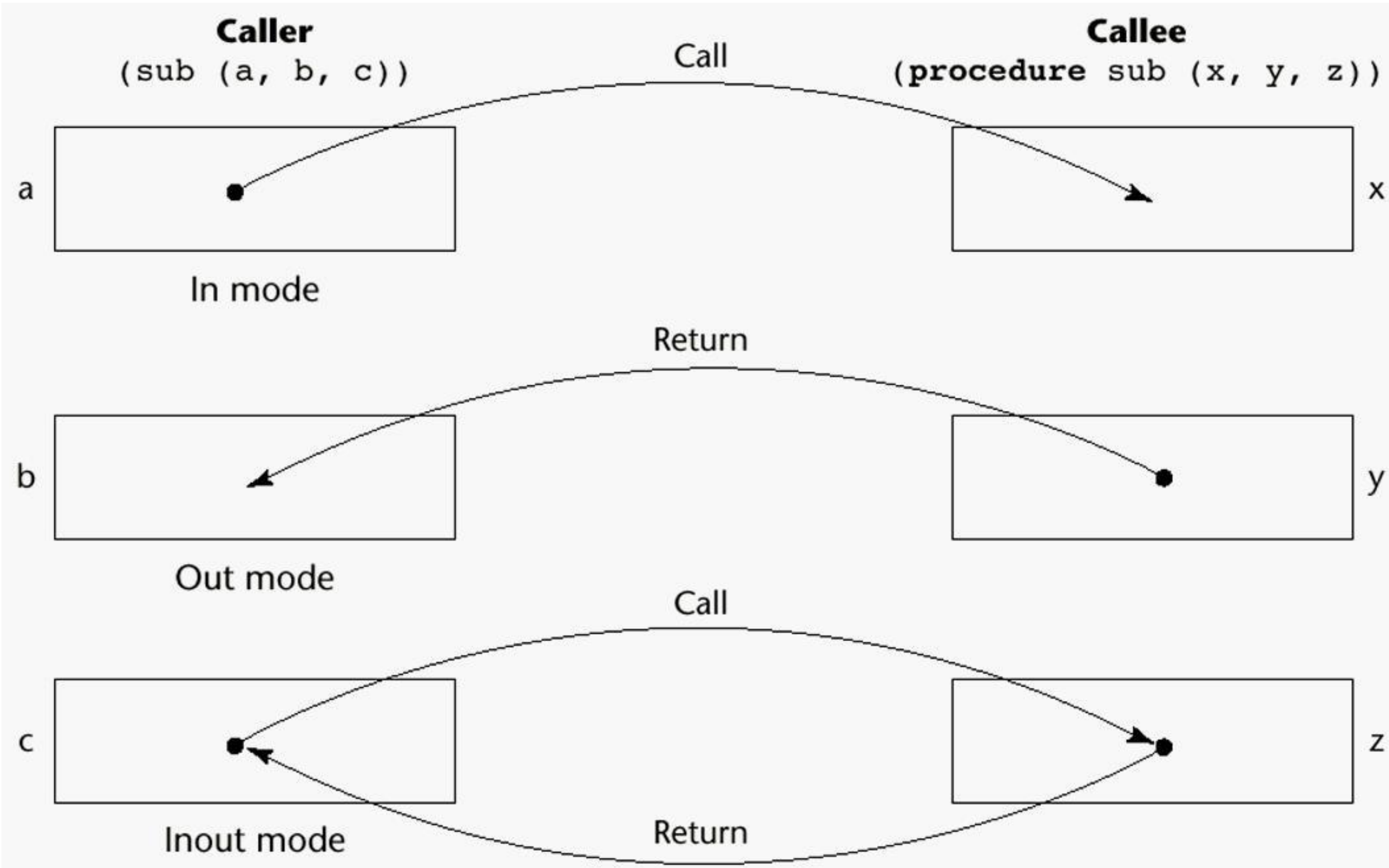
Semantic Models of Parameter Passing

In mode

Out mode

Inout mode

Models of Parameter Passing



Conceptual Models of Transfer

Physically move a path

Move an access path

Pass-by-Value (In Mode)

The value of the actual parameter is used to initialize the corresponding formal parameter

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

Pass-by-Result (Out Mode)

When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move

– Require extra storage location and copy operation

Potential problem: `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of p1

Pass-by-Value-Result (inout Mode)

A combination of pass-by-value and pass-by-result

Sometimes called pass-by-copy

Formal parameters have local storage

Disadvantages:

- Those of pass-by-result
- Those of pass-by-value

Pass-by-Reference (Inout Mode)

Pass an access path

Also called pass-by-sharing

Advantage: Passing process is efficient (no copying and no duplicated storage)

Disadvantages

- Slower accesses (compared to pass-by-value) to formal parameters
- Potentials for unwanted side effects (collisions)
- Unwanted aliases (access broadened)

Pass-by-Name (Inout Mode)

By textual substitution

Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment

Allows flexibility in late binding

Implementing Parameter-Passing Methods

In most language parameter communication takes place thru the run-time stack

Pass-by-reference are the simplest to implement; only an address is placed in the stack

A subtle but fatal error can occur with pass-by-reference and pass-by-value-result: a formal parameter corresponding to a constant can mistakenly be changed

Parameter Passing Methods of Major Languages

C

- Pass-by-value
- Pass-by-reference is achieved by using pointers as parameters

C++

- A special pointer type called reference type for pass-by-reference

Java

- All parameters are passed are passed by value
- Object parameters are passed by reference

Ada

- Three semantics modes of parameter transmission: `in, out, in out; in` is the default mode
- Formal parameters declared `out` can be assigned

but not referenced; those declared `in` can be

Parameter Passing Methods of Major Languages (continued)

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

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal, FORTRAN 90, Java, and Ada: it is always required
- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

Multidimensional Arrays as Parameters

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

Multidimensional Arrays as Parameters: C and C++

Programmer is required to include the declared sizes of all but the first subscript in the actual parameter

Disallows writing flexible subprograms

Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

Multidimensional Arrays as Parameters: Ada

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

Multidimensional Arrays as Parameters: Fortran

Formal parameter that are arrays have a declaration after the header

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

Multidimensional Arrays as Parameters: Java and C#

Similar to Ada

Arrays are objects; they are all single-dimensioned, but the elements can be arrays

Each array inherits a named constant (`length` in Java, `Length` in C#) that is set to the length of the array when the array object is created

Design Considerations for Parameter Passing

Two important considerations

- Efficiency
- One-way or two-way data transfer

But the above considerations are in conflict

- Good programming suggest limited access to variables, which means one-way whenever possible
- But pass-by-reference is more efficient to pass structures of significant size

Parameters that are Subprogram Names

It is sometimes convenient to pass subprogram names as parameters

Issues:

- Are parameter types checked?
- What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Parameter Type Checking

C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked

FORTRAN 95 type checks

Ada does not allow subprogram parameters; an alternative is provided via Ada's generic facility

Java does not allow method names to be passed as parameters

Parameters that are Subprogram Names: Referencing Environment

Shallow binding: The environment of the call statement that enacts the passed subprogram
- Most natural for dynamic-scoped languages

Deep binding: The environment of the definition of the passed subprogram
- Most natural for static-scoped languages

Ad hoc binding: The environment of the call statement that passed the subprogram

Overloaded Subprograms

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

Generic Subprograms

A *generic* or *polymorphic subprogram* takes parameters of different types on different activations

Overloaded subprograms provide *ad hoc polymorphism*

A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*

- A cheap compile-time substitute for dynamic binding

Generic Subprograms (continued)

Ada

- Versions of a generic subprogram are created by the compiler when explicitly instantiated by a declaration statement
- Generic subprograms are preceded by a `generic` clause that lists the generic variables, which can be types or other subprograms

Generic Subprograms (continued)

C++

- Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
- Generic subprograms are preceded by a `template` clause that lists the generic variables, which can be type names or class names

Generic Subprograms (continued)

Java 5.0

- Differences between generics in Java 5.0 and those of C++ and Ada:

1. Generic parameters in Java 5.0 must be classes

Java 5.0 generic methods are instantiated just once as truly generic methods

3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters 4. Wildcard types of generic parameters

Generic Subprograms (continued)

C# 2005

Supports generic methods that are similar to those of Java 5.0

One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type

Examples of parametric polymorphism: C++

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

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

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

Design Issues for Functions

Are side effects allowed?

- Parameters should always be in-mode to reduce side effect (like Ada)

What types of return values are allowed?

- Most imperative languages restrict the return types
- C allows any type except arrays and functions
- C++ is like C but also allows user-defined types
- Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
- Java and C# methods can return any type (but because methods are not types, they cannot be returned)
- Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
- Lua allows functions to return multiple values

User-Defined Overloaded Operators

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

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

Coroutines

A *coroutine* is a subprogram that has multiple entries and controls them itself – supported directly in Lua

Also called *symmetric control:* caller and called coroutines are on a more equal basis

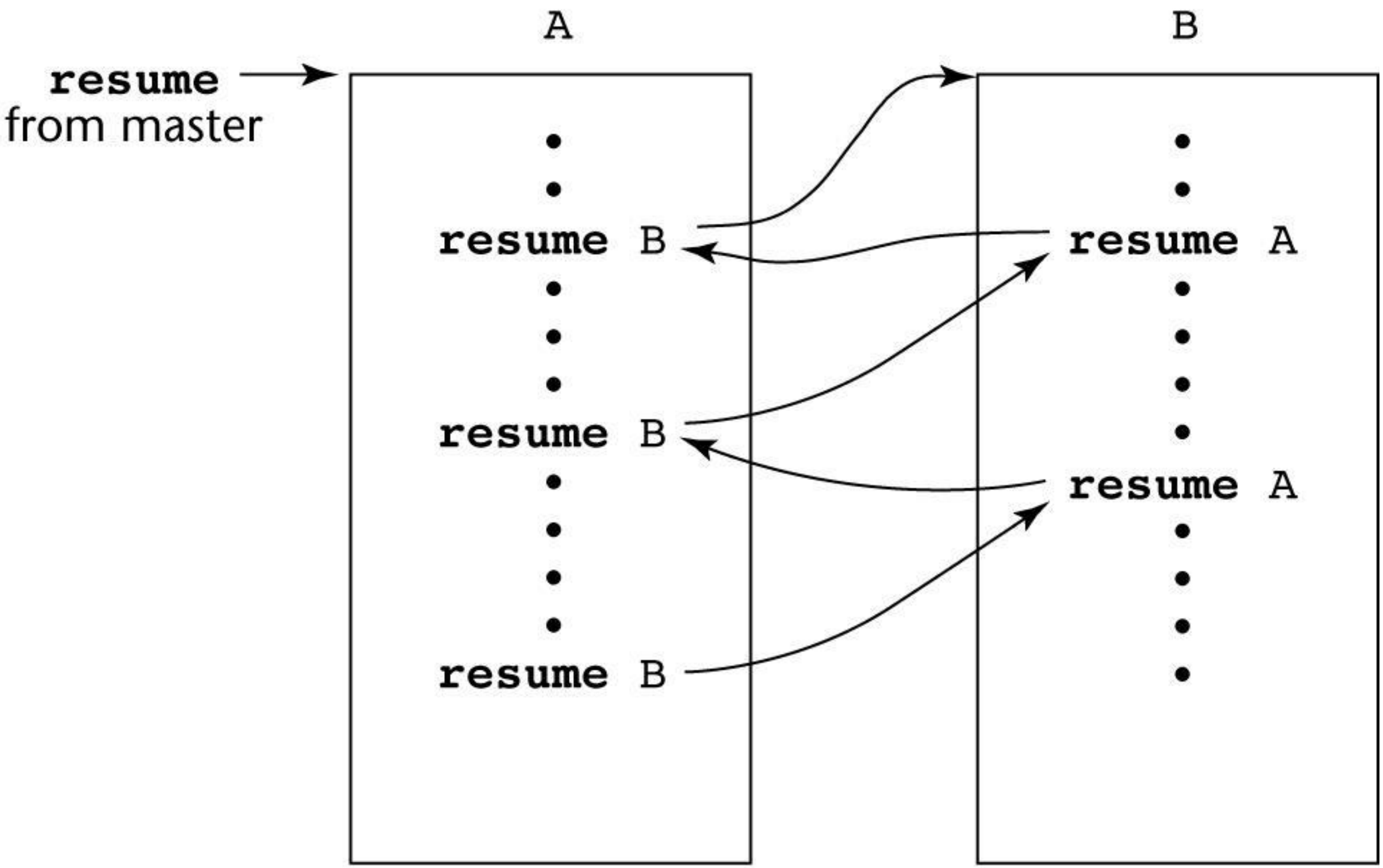
A coroutine call is named a *resume*

The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine

Coroutines repeatedly resume each other, possibly forever

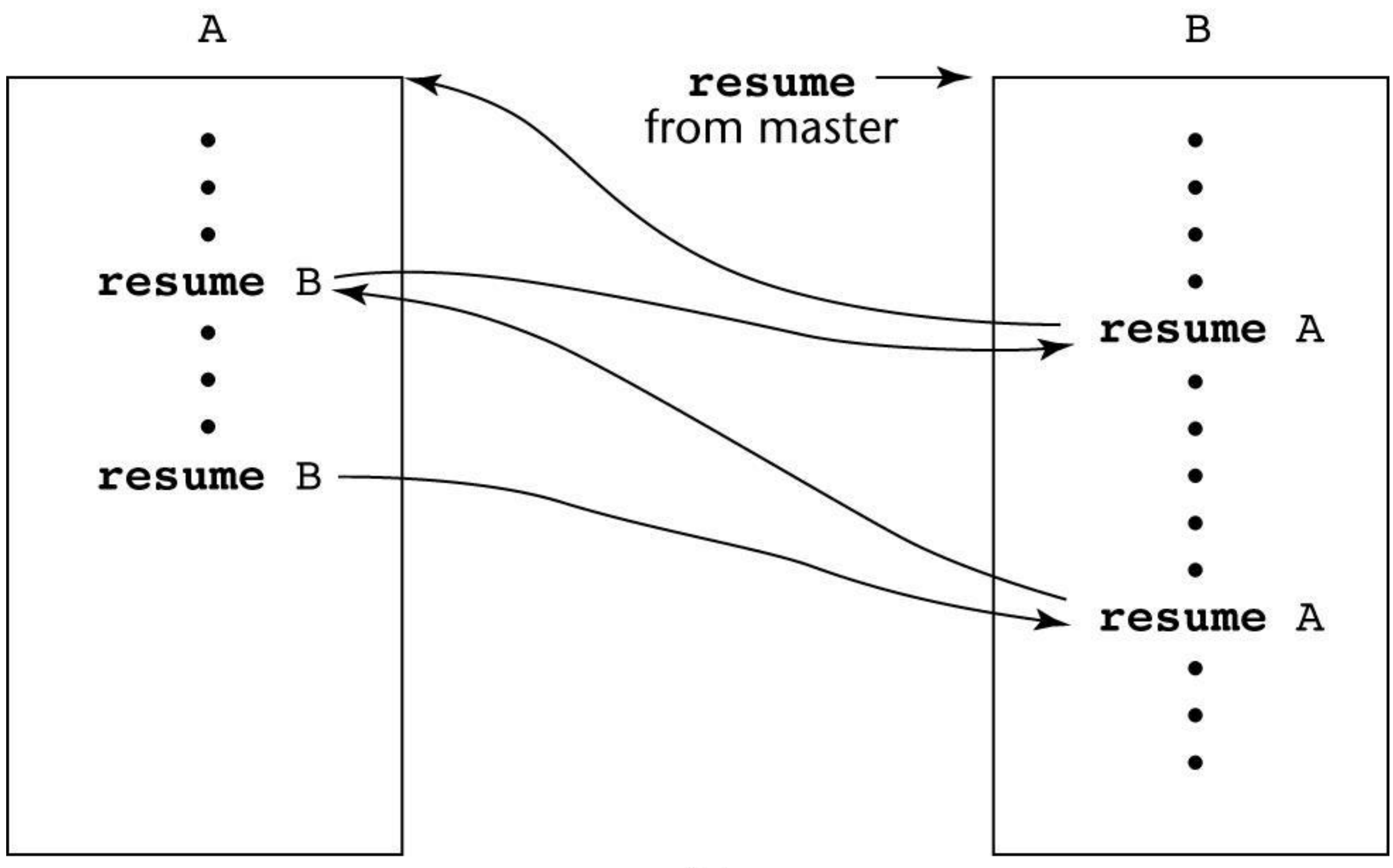
Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

Coroutines Illustrated: Possible Execution Controls



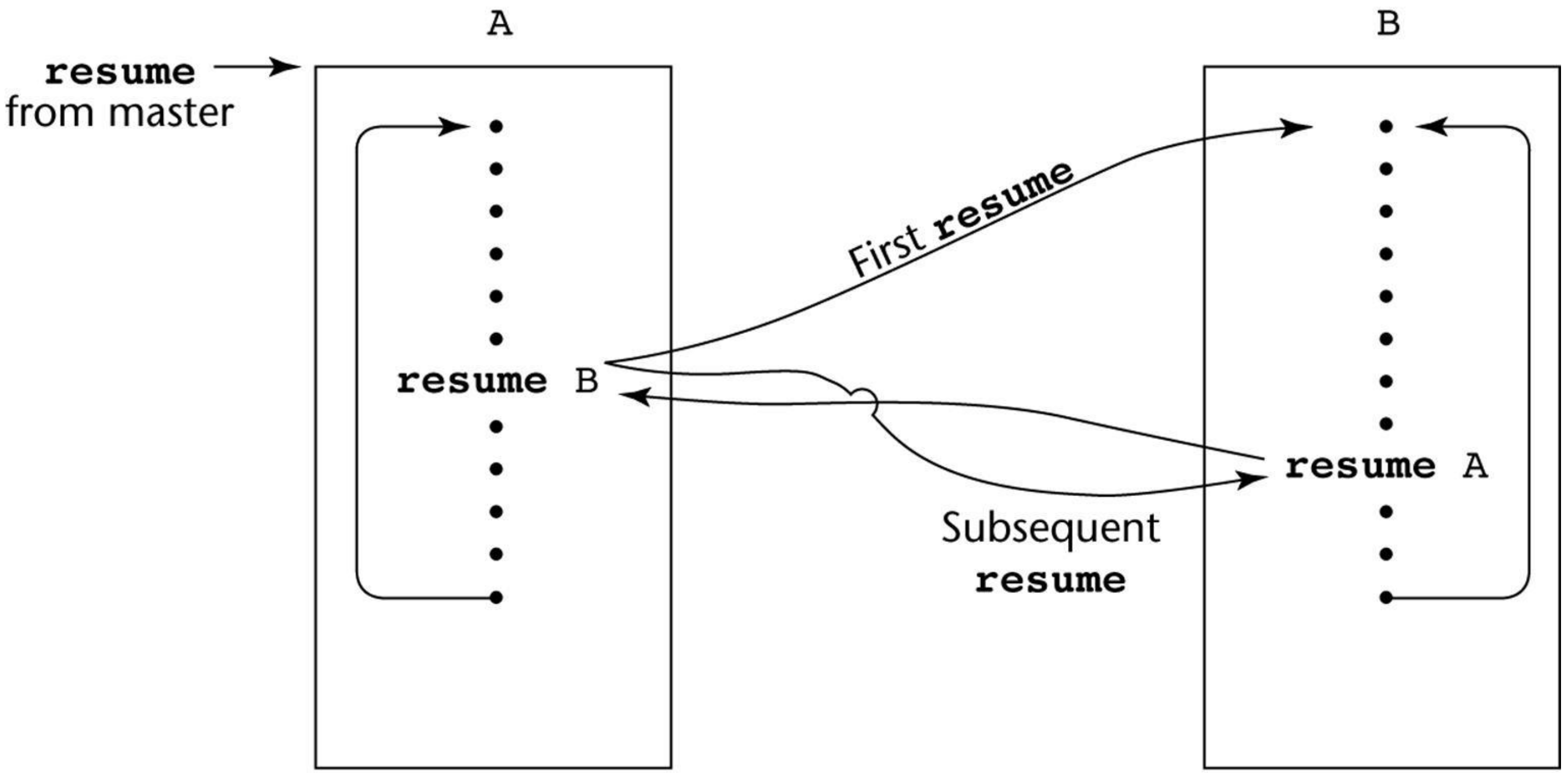
(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops



The General Semantics of Calls and Returns

The subprogram call and return operations of a language are together called its *subprogram linkage*

General semantics of subprogram calls

- Parameter passing methods
- Stack-dynamic allocation of local variables
- Save the execution status of calling program
- Transfer of control and arrange for the return
- If subprogram nesting is supported, access to nonlocal variables must be arranged

The General Semantics of Calls and Returns

General semantics of subprogram returns:

- In mode and inout mode parameters must have their values returned
- Deallocation of stack-dynamic locals
- Restore the execution status
- Return control to the caller

Implementing "Simple" Subprograms: Call Semantics

Call Semantics:

Save the execution status of the caller
Pass the parameters
Pass the return address to the callee
Transfer control to the callee

Implementing "Simple" Subprograms: Return Semantics

Return Semantics:

- If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
- If it is a function, move the functional value to a place the caller can get it
- Restore the execution status of the caller
- Transfer control back to the caller

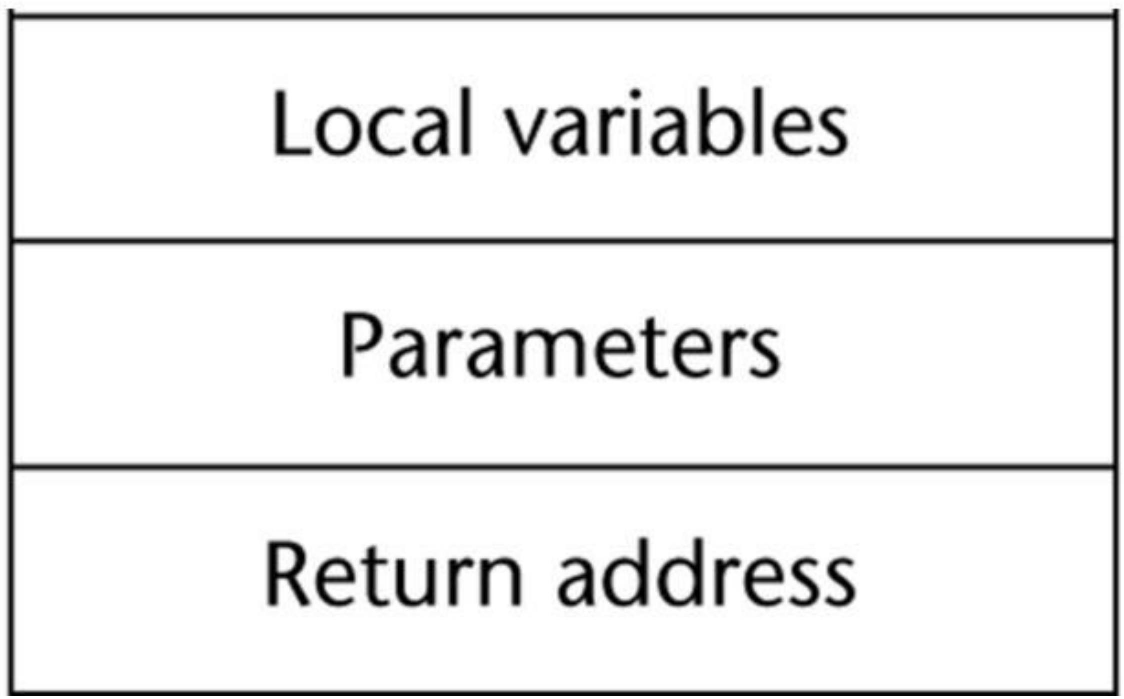
Required storage:

- Status information, parameters, return address, return value for functions

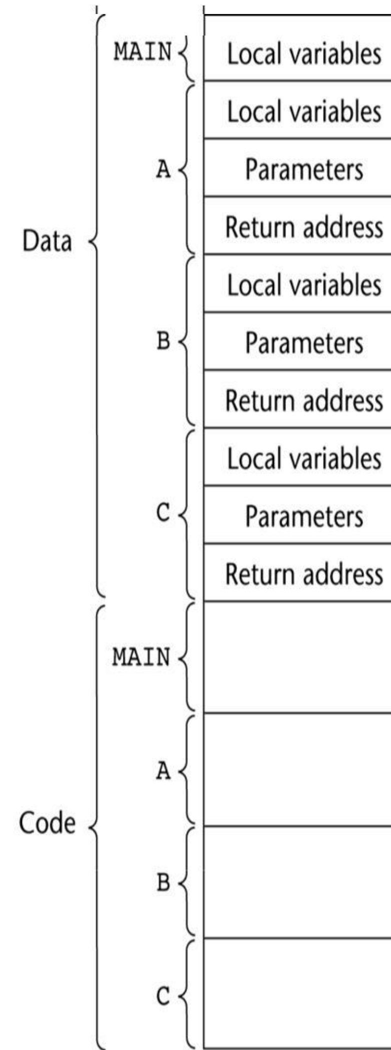
Implementing "Simple" Subprograms: Parts

- Two separate parts: the actual code and the non-code part (local variables and data that can change)
- The format, or layout, of the non-code part of an executing subprogram is called an *activation record*
- An *activation record instance* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

An Activation Record for "Simple" Subprograms



Code and Activation Records of a Program with "Simple" Subprograms

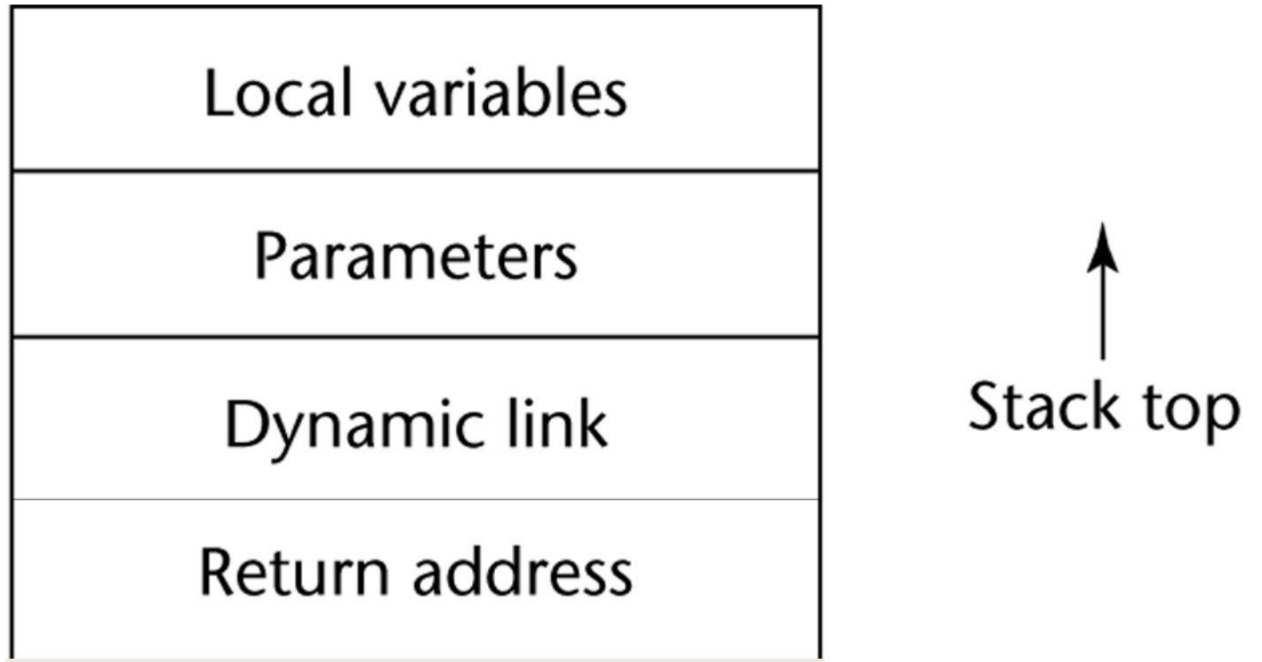


Implementing Subprograms with Stack-Dynamic Local Variables

More complex activation record

- The compiler must generate code to cause implicit allocation and deallocation of local variables
- Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

Typical Activation Record for a Language with Stack-Dynamic Local Variables

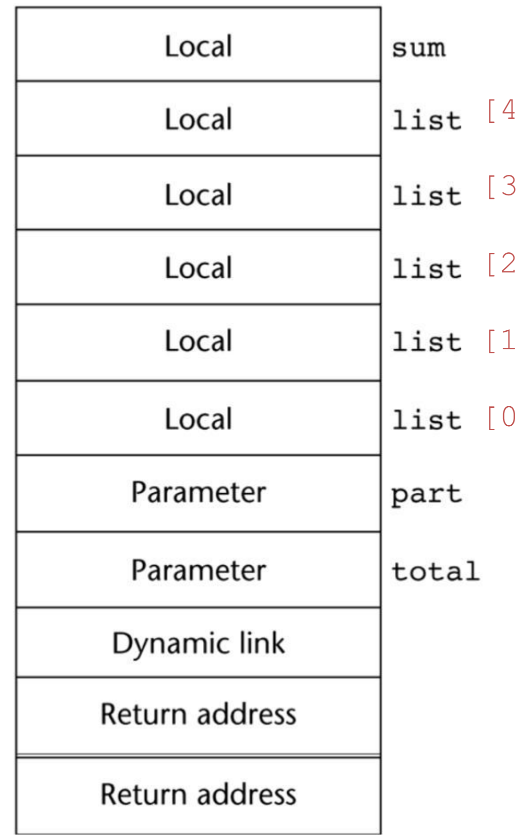


Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record format is static, but its size may be dynamic
- The *dynamic link* points to the top of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called
- Activation record instances reside on the run-time stack
- The *Environment Pointer* (EP) must be maintained by the run-time system. It always points at the base of the activation record instance of the currently executing program unit

An Example: C Function

```
void sub(float total, int part)
{
   int list[5];
   float sum;
   …
}
```

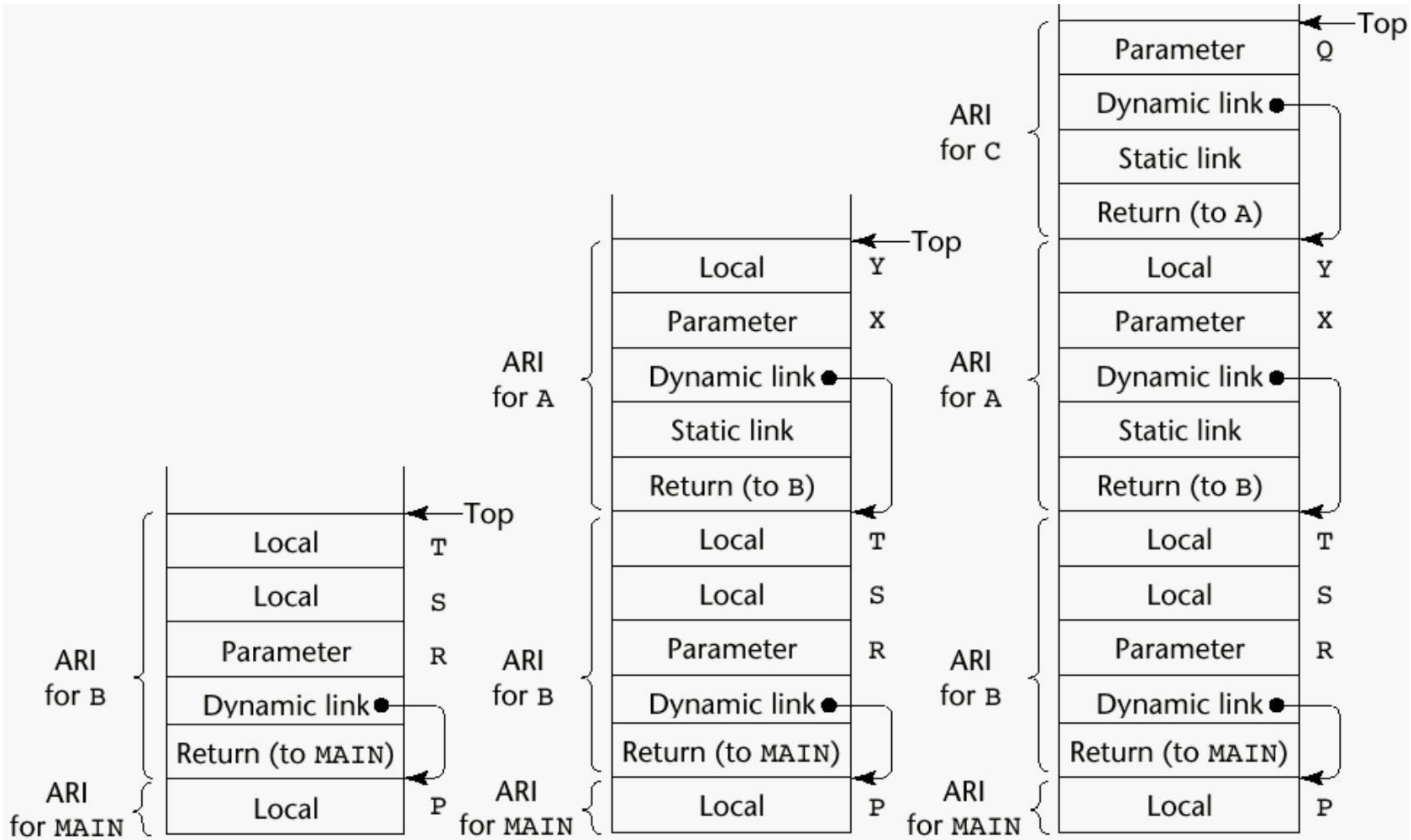


An Example Without Recursion

```
void A(int x) {
    int y;
    ...
    C(y);
    ...
}
void B(float r) {
    int s, t;
    ...
    A(s);
    ...
}
void C(int q) {
    ...
}
void main() {
    float p;
    ...
    B(p);
    ...
}
```

main calls B
B calls A
A calls C

An Example Without Recursion



ARI = activation record instance

Dynamic Chain and Local Offset

The collection of dynamic links in the stack at a given time is called the *dynamic chain*, or *call chain*

Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the *local_offset*

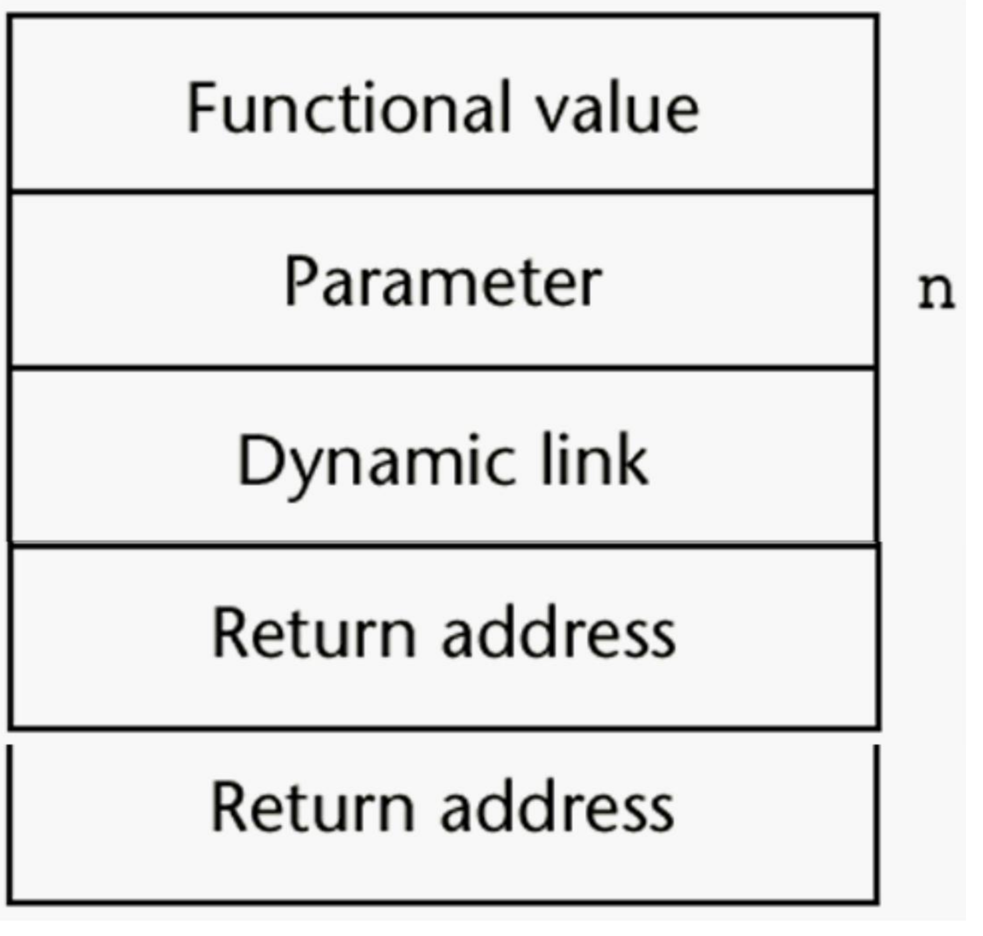
The local_offset of a local variable can be determined by the compiler at compile time

An Example With Recursion

The activation record used in the previous example supports recursion, e.g.

```
int factorial (int n) {
    <-----------------------------1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
    <----------------------------- 2
}
void main() {
  int value;
  value = factorial(3);
    <----------------------------- 3
}
```

Activation Record for `factorial`



Nested Subprograms

Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, Python, JavaScript, Ruby, and Lua) use stack-dynamic local variables and allow subprograms to be nested

All variables that can be non-locally accessed reside in some activation record instance in the stack

The process of locating a non-local reference:

- Find the correct activation record instance
- Determine the correct offset within that activation record instance

Locating a Non-local Reference

- Finding the offset is easy
- Finding the correct activation record instance
  – Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made

Static Scoping

A *static chain* is a chain of static links that connects certain activation record instances

The *static link* in an activation record instance for subprogram A points to one of the activation record instances of A's static parent

The static chain from an activation record instance connects it to all of its static ancestors

Static_depth is an integer associated with a static scope whose value is the depth of nesting of that scope

Static Scoping (continued)

The *chain_offset* or *nesting_depth* of a nonlocal reference is the difference between the static_depth of the reference and that of the scope when it is declared

A reference to a variable can be represented by the pair: (chain_offset, local_offset),

where local_offset is the offset in the activation record of the variable being referenced

Example Ada Program

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
      A := B + C;  <-----------------------1
    end;   -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
        Sub1;
        E := B + A:   <--------------------2
        end; -- of Sub3
      begin -- of Sub2
      Sub3;
      A := D + E;  <-----------------------3
      end; -- of Sub2 }
    begin -- of Bigsub
     Sub2(7);
    end; -- of Bigsub
  begin
  Bigsub;
end; of Main 2 }
```

Example Ada Program (continued)

Call sequence for `Main_2`

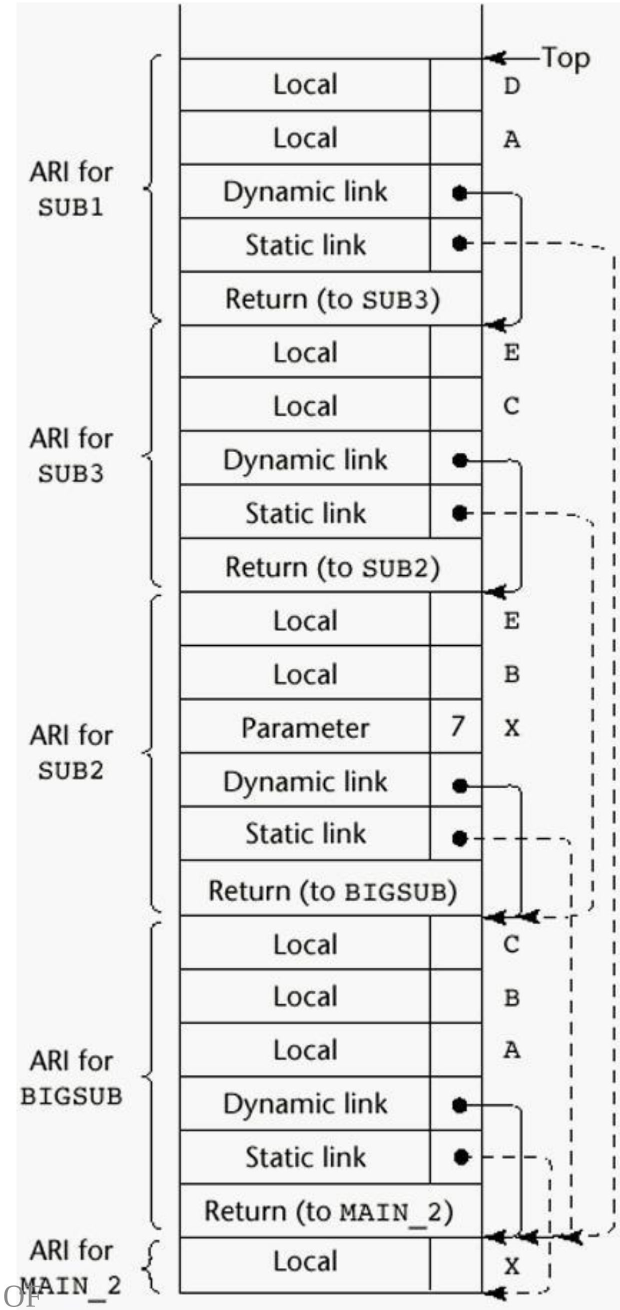
`Main_2` **calls** `Bigsub`

`Bigsub` **calls** `Sub2`

`Sub2` **calls** `Sub3`

`Sub3` **calls** `Sub1`

Stack Contents at Position 1



Static Chain Maintenance

At the call,

The activation record instance must be built

The dynamic link is just the old stack top pointer

The static link must point to the most recent ari of the static parent

Two methods:

Search the dynamic chain

Treat subprogram calls and definitions like variable references and definitions

Evaluation of Static Chains

Problems:

- A nonlocal areference is slow if the nesting depth is large
- Time-critical code is difficult:
  - Costs of nonlocal references are difficult to determine
  - Code changes can change the nesting depth, and therefore the cost

Displays

An alternative to static chains that solves the problems with that approach

Static links are stored in a single array called a display

The contents of the display at any given time is a list of addresses of the accessible activation record instances

Blocks

Blocks are user-specified local scopes for variables

An example in C

```
{int temp;
  temp = list [upper];
  list [upper] = list [lower];
  list [lower] = temp
}
```

The lifetime of `temp` in the above example begins when control enters the block

An advantage of using a local variable like `temp` is that it cannot interfere with any other variable with the same name

Implementing Blocks

Two Methods:

Treat blocks as parameter-less subprograms that are always called from the same location

– Every block has an activation record; an instance is created every time the block is executed

Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Dynamic Scoping

Deep Access: non-local references are found by searching the activation record instances on the dynamic chain

- Length of the chain cannot be statically determined

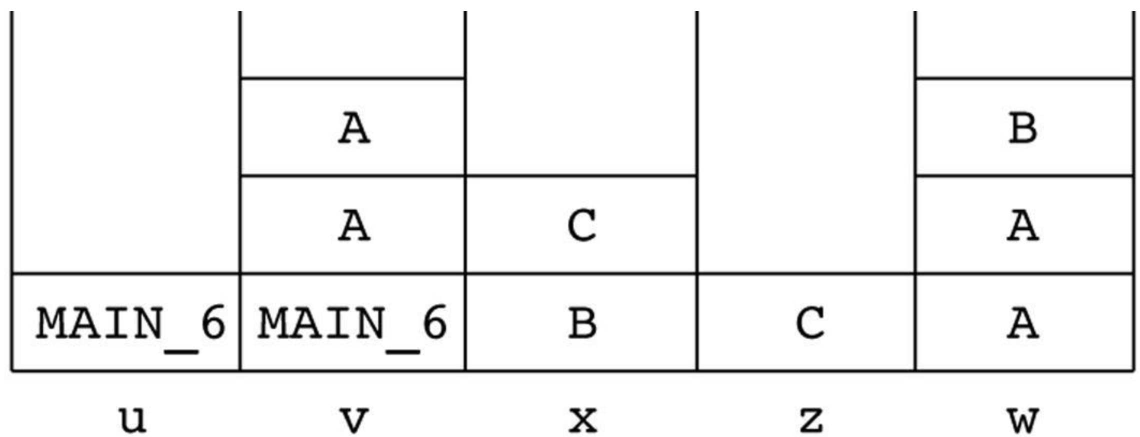
  Every activation record instance must have variable names

Shallow Access: put locals in a central place

- One stack for each variable name
- Central table with an entry for each variable name

Using Shallow Access to Implement Dynamic Scoping

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```



(The names in the stack cells indicate the program units of the variable declaration.)

Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A coroutine is a special subprogram with multiple entries

Summary

- Subprogram linkage semantics requires many action by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex
- Subprograms with stack-dynamic local variables and nested subprograms have two components
  – actual code
  – activation record

Summary

- Activation record instances contain formal parameters and local variables among other things
- Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or thru some central variable table method

Unit-4

Abstract Data Types

Concurrency

Exception Handling

Logic Programming Language

CONCEPTS

Abstract Data types

Concurrency

Exception Handling

Logic Programming Language

CONCEPTS

Introduction to logic programming language A Brief Introduction to Predicate Calculus Predicate Calculus and Proving Theorems An Overview of Logic Programming The Origins of Prolog

The Basic Elements of Prolog Deficiencies of Prolog Applications of Logic Programming

Abstract Data types

An *abstraction* is a view or representation of an entity that includes only the most significant attributes.

The concept of *abstraction* is fundamental in programming (and computer science).

Nearly all programming languages support *process abstraction* with subprograms.

Nearly all programming languages designed since 1980 support *data abstraction.*

Introduction to Data Abstraction

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

–The representation of, and operations on, objects of the type are defined in a single syntactic unit.

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

Encapsulation

Original motivation :

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

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

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

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

Language Requirements for Data Abstraction

A syntactic unit in which to encapsulate the type definition.

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

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

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

e.g., iterators, constructors, destructors.

Language Design Issues

Encapsulate a single type, or something more?

What types can be abstract?

Can abstract types be parameterized?

What access controls are provided?

Language Examples

1. Simula 67

Provided encapsulation, but no information Hiding.

2. Ada

The encapsulation construct is the package

Packages usually have two parts:

Specification package (the interface)

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

Evaluation of Ada Abstract Data Types

Lack of restriction to pointers is better - Cost is recompilation of clients when the representation is changed.

Cannot import specific entities from other Packages.

Parameterized Abstract Data Types

1. Ada Generic Packages

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

---> SHOW GENERIC_STACK package and two instantiations .

C++ Templated Classes

Classes can be somewhat generic by writing parameterized constructor functions.

```
stack (int size) {
stk_ptr = new int [size];
max_len = size - 1;
top = -1;
}
stack (100) stk;
```

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

---> SHOW the templated class stack .

- Java does not support generic abstract data types

Object Oriented Programming in Smalltalk

Type Checking and Polymorphism:

All bindings of messages to methods is dynamic.

The process is to search the object to which the message is sent for the method; if not found, search the superclass, etc.

Because all variables are typeless, methods are all polymorphic

Inheritance.

All subclasses are subtypes (nothing can be hidden).

All inheritance is implementation inheritance.

No multiple inheritance.

Methods can be redefined, but the two are not related.

C++

General Characteristics:

Mixed typing system.

Constructors and destructors.

Elaborate access controls to class entities.

Inheritance:

A class need not be subclasses of any class.

Access controls for members are:

Private (visible only in the class and friends).

Public (visible in subclasses and clients).

Protected (visible in the class and in subclasses).

- In addition, the subclassing process can be declared with access controls, which define potential changes in access by subclasses.
- Multiple inheritance is supported

Java

Dynamic Binding

In Java, all messages are dynamically bound to methods, unless the method is final.

Encapsulation

Two constructs, classes and packages.

Packages provide a container for classes that are related.

Entities defined without an scope (access) modifier have package scope, which makes them visible throughout the package in which they are defined

Every class in a package is a friend to the package scope entities elsewhere in the package.

Ada 95

Example:

```
with PERSON_PKG; use PERSON_PKG;
package STUDENT_PKG is
type STUDENT is new PERSON with
record
GRADE_POINT_AVERAGE : FLOAT;
GRADE_LEVEL : INTEGER;
end record;
procedure DISPLAY (ST: in STUDENT);
end STUDENT_PKG;
```

DISPLAY is being overriden from PERSON_PKG

All subclasses are subtypes

Single inheritance only, except through generics

Concurrency

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

Categories of Concurrency:

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

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

- Coroutines provide only quasiconcurrency.

Reasons to Study Concurrency

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

Computers capable of physical concurrency are now widely used.

Design Issues for Concurrency

How is cooperation synchronization provided?

How is competition synchronization provided?

How and when do tasks begin and end execution?

Are tasks statically or dynamically created?

Methods of Providing Synchronization

Semaphores

Monitors

Message Passing

Semaphores

Semaphores (Dijkstra - 1965).

A semaphore is a data structure consisting of a counter and a queue for storing task descriptors.

Semaphores can be used to implement guards on the code that accesses shared data structures.

Semaphores have only two operations, wait and release (originally called P and V by Dijkstra).

Semaphores can be used to provide both competition and cooperation synchronization

Example

wait(aSemaphore)

if aSemaphore's counter > 0 then
Decrement aSemaphore's counter

else

Put the caller in aSemaphore's queue

Attempt to transfer control to
some ready task

(If the task ready queue is empty,

deadlock
occurs) end

Example

release(aSemaphore)

if aSemaphore’s queue is empty then
Increment aSemaphore’s counter

else

Put the calling task in the task
ready queue

Transfer control to a task
from aSemaphore’s queue

end

Monitors

Competition Synchronization with Monitors:

Access to the shared data in the monitor is

limited by the implementation to a single process at a time; therefore, mutually exclusive access is inherent in the semantic definition of the monitor.

- Multiple calls are queued.

Monitors

Cooperation Synchronization with Monitors:

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

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

Differs from send because delay always blocks the caller.

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

-It also takes a process from the parameter.

-queue (if the queue isn't empty) and starts it.

-Differs from release because it always has some effect (release does nothing if the queue is empty).

Message Passing

Competition Synchronization with Message Passing:

Example:

a shared buffer.

Encapsulate the buffer and its operations in a task.

Competition synchronization is implicit in the semantics of accept clauses.

Only one accept clause in a task can be active at any given time.

Java Threads

Competition Synchronization with Java Threads:

A method that includes the synchronized modifier disallows any other method from running on the object while it is in execution.

If only a part of a method must be run without interference, it can be synchronized.

Cooperation Synchronization with Java Threads:

The wait and notify methods are defined in Object, which is the root class in Java, so all objects inherit them.

The wait method must be called in a loop.

Example - the queue.

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.

Design Issues for Exception Handling

How and where are exception handlers specified and what is their scope?

How is an exception occurrence bound to an exception handler?

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 their own?

Can built-in exceptions be explicitly raised?

Are hardware-detectable errors treated as exceptions that can be handled?

Are there any built-in exceptions?

How can exceptions be disabled, if at all?

Ada Exception Handling

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

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

Handler form:

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

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

Binding Exceptions to Handlers

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

Procedures - propagate it to the caller.

Blocks - propagate it to the scope in which it occurs.

Package body - propagate it to the declaration part of the unit that declared the package (if it is a library unit (no static parent), the program is terminated).

Task - no propagation; if it has no handler, execute it; in either case, mark it "completed“.

C++ Exception Handling

try {

code that is expected to raise an exception} catch (formal parameter) {

handler code

}.....

catch (formal parameter) {

handler code

}

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

The formal parameter need not have a variable.

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

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

Java Exception Handling

The finally Clause:

Can appear at the end of a try construct

Form:

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

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

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

```
for (index = 0; index < 100; index++) {
…
if (…) {
return;
}
```

Evaluation

The types of exceptions makes more sense than in the case of C++.

The throws clause is better than that of C++

(The throw clause in C++ says little to the programmer).

The finally clause is often useful.

The Java interpreter throws a variety of exceptions that can be handled by user programs.

Introduction to logic programming

Logic programming languages, sometimes called *declarative* programming Languages.

Express programs in a form of symbolic logic.

Use a logical inferencing process to produce results.

Declarative rather that procedural:

–Only specification of *results* are stated (not detailed *procedures* for producing them).

Proposition:

A logical statement that may or may not be true.

–Consists of objects and relationships of objects to each other.

Symbolic Logic:

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

–Express propositions.

–Express relationships between propositions.

–Describe how new propositions can be inferred from other propositions.
(Particular form of symbolic logic used for logic programming called *predicate Calculus)*

Object Representation

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

Constant: a symbol that represents an object.

Variable: a symbol that can represent different objects at different times.

–Different from variables in imperative languages.

Compound Terms:

Atomic propositions consist of compound terms.

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

–Mathematical function is a mapping.

–Can be written as a table.

Parts of a Compound Term:

Compound term composed of two parts:

Example

Functor: function symbol that names the relationship.

–Ordered list of parameters (tuple).

Examples:

student(jon)

like(seth, OSX)

like(nick, windows)

like(jim, linux)

Forms of a Proposition

Propositions can be stated in two forms:

–*Fact*: proposition is assumed to be true.

–*Query*: truth of proposition is to be determined.

Compound proposition:

–Have two or more atomic propositions.

–Propositions are connected by operators.

Clausal Form

Too many ways to state the same thing

-Use a standard form for propositions.

Clausal form:

–B1 B2 … Bn A1 A2 … Am

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

Antecedent: right side.

Consequent: left side.

Predicate Calculus and Proving Theorems

-use of propositions is to discover new theorems that can be inferred from known axioms and theorems.

Resolution: an inference principle that allows inferred propositions to be computed from given propositions r*esolution.*

Unification: finding values for variables in propositions that allows matching process to succeed.

Instantiation: assigning temporary values to variables to allow unification to succeed after instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value.

Theorem Proving

-Basis for logic programming.

-When propositions used for resolution, only restricted form can be used.

Horn clause - can have only two forms.

–*Headed*: single atomic proposition on left side.

–*Headless*: empty left side (used to state facts).

-Most propositions can be stated as Horn clauses.

Basic Elements of Prolog

Terms:

-Edinburgh Syntax.

Term: a constant, variable, or structure.

Constant: an atom or an integer.

Atom: symbolic value of Prolog.

Atom consists of either:

–a string of letters, digits, and underscores beginning with a lowercase letter.

–a string of printable ASCII characters delimited by apostrophes.

Terms: Variables and Structures

-*Variable*: any string of letters, digits, and underscores beginning with an uppercase letter.

-*Instantiation*: binding of a variable to a value.

–Lasts only as long as it takes to satisfy one complete goal.

-*Structure*: represents atomic proposition

functor(*parameter list*).

Fact Statements

-Used for the hypotheses.

-Headless Horn clauses:

```
female(shelley).
male(bill).
father(bill, jake).
```

Rule Statements

-Used for the hypotheses.

-Headed Horn clause:

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

–May be single term or conjunction.

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

–Must be single term.

Conjunction: multiple terms separated by logical AND operations (implied)

Example Rules:

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

Can use variables (*universal objects*) to generalize meaning:

parent(X,Y):- mother(X,Y).

parent(X,Y):- father(X,Y).

grandparent(X,Z):- parent(X,Y), parent(Y,Z).

sibling(X,Y):- mother(M,X), mother(M,Y),
father(F,X), father(F,Y).

Goal Statements

-For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement.*

-Same format as headless Horn **eg:** man(fred)

-Conjunctive propositions and propositions with variables also legal goals.

eg: father(X,mike)

Inferencing Process of Prolog

-Queries are called goals.

-If a goal is a compound proposition, each of the facts is a subgoal.

-To prove a goal is true, must find a chain of inference rules and/or facts.

For goal Q:

:- A

:- B

…

:- P

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

Simple Arithmetic

-Prolog supports integer variables and integer arithmetic.

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

eg: A is B / 17 + C

-Not the same as an assignment statement!

Example: speed(ford,100).

speed(chevy,105).

speed(dodge,95).

speed(volvo,80).

time(ford,20).

time(chevy,21).

time(dodge,24).

time(volvo,24).

distance(X,Y) :- speed(X,Speed),

time(X,Time),

Y is Speed * Time.

Trace

-Built-in structure that displays instantiations at each step.

-*Tracing model* of execution - four events:

–*Call* (beginning of attempt to satisfy goal).

–*Exit* (when a goal has been satisfied).

–*Redo* (when backtrack occurs).

–*Fail* (when goal fails).

Example

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

Bindings and scope

A PROLOG program consists of one or more relations.

The scope of every relation is the entire program.

It is not possible in PROLOG to define a relation locally to another relation, nor to group relations into packages.

Control

In principle, the order in which resolution is done should not affect the set of answers yielded by a query (although it will affect the order in which these answers are found).

In practical logic programming, however, the order is very important

Deficiencies of prolog

Resolution order control

Closed word assumption: When an assertion is tested, therefore, success means true and failure means either unknown or false. As this is rather inconvenient, PROLOG bends the rules of logic by ignoring the distinction between unknown and false. In other words, an assertion is assumed to be false if it cannot be inferred to be true. This is called the ***closed world assumption***

Negation problem.

Applications of Logic Programming

Relational database management system:

RDBMS stores data in the form of tables and queries.

Prolog can replace the DML,DDL and query language which are implanted in imperative languages.

Expert Systems

Expert systems consists of database of facts, an inferencing process, a human interface to look like an expert human consultant.

Logical programming helps to solve the incompleteness of database.

Applications of logic programming(cont..)

Natural language processing

Few kinds of natural processing languages can be done using logical programming

Unit-5

Functional Programming Languages

Scripting Language

CONCEPTS

Introduction

Fundamentals of FPL

LISP

ML

HASKELL

Applications of FPL

Scripting languages

FUNTIONAL PROGRAMMING LANGUAGE

The design of the imperative languages is based directly on Von Nuemann Architechture.

The design of the functional language is based on mathematical functions.

MATHEMATICAL FUNCTION

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 l(x) x * x * x

For the function cube (x) = x * x * x

Lambda expressions describe nameless functions

Mathematical function(cont..)

Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

e.g. (l(x) x * x * x)(3) which evaluates to 27

A Function for Constructing Functions

DEFINE - Two forms:

To bind a symbol to an expression

e.g.,

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

Fundamentals of Functional Programming Languages

The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible.

The basic process of computation is fundamentally different in a FPL than in an imperative language.

In an imperative language, operations are done and the results are stored in variables for later use

Fundamentals of FPL(cont..)

Management of variables is a constant concern and source of complexity for imperative programming.

In an FPL, variables are not necessary, as is the case in mathematics.

In an FPL, the evaluation of a function always produces the same result given the same parameters.

This is called *referential transparency.*

LISP

The first functional programming language.

Data object types: originally only atoms and lists.

List form: parenthesized collections of sublists and/or atoms

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

A Bit of LISP

Originally, LISP was a typeless language.

There were only two data types, atom and list.

LISP lists are stored internally as single-linked lists.

Lambda notation is used to specify functions and function definitions, function applications,and data all have the same form.

INTRODUCTION TO SCHEME

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

Uses only static scoping.

Functions are first-class entities.

-They can be the values of expressions and elements of lists

They can be assigned to variables and passed as parameters

Primitive Functions:

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

e.g., (+ 5 2) yields 7

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

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

QUOTE

QUOTE can be abbreviated with the apostrophe prefix operator

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

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

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

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

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

CONS takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result e.g., (CONS 'A '(B C)) returns (A B C)

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

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

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

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

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

5. *Output Utility Functions:*

(DISPLAY expression)

(NEWLINE)

Lambda Expressions:

Form is based on l notation e.g.,
(LAMBDA (L) (CAR (CAR L))) L is
called a bound variable Lambda
expressions can be applied
e.g., ((LAMBDA (L) (CAR (CAR L))) '((A B) C D))

To bind names to lambda expressions

e.g.,(DEFINE (cube x) (* x x x))

Example use:(cube 4)

- 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.

  The value of the last expression in the body is the value of the function.

Control Flow:

Selection- the special form, IF

(IF predicate then_exp else_exp) e.g.,(IF (<> count 0)

(/ sum
count) 0
)

Multiple Selection - the special form, COND

- General form:
- (COND

```
 (predicate_1 expr {expr})
 (predicate_1 expr {expr})
   ...
(predicate_1 expr {expr})
(ELSE expr {expr})
)
```

Returns the value of the last expr in the first pair whose predicate evaluates to true

COMMON LISP

A combination of many of the features of the popular dialects of LISP around in the early 1980s.

A large and complex language--the opposite of Scheme.

Includes: records, arrays, Complex numbers, character strings, powerful i/o capabilities, packages with access control, imperative features like those of Scheme ,iterative control statements.

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

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

ML(cont..)

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

Includes lists and list operations

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

Function declaration form:

fun function_name (formal_parameters)
= function_body_expression;

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

ML(cont..)

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

Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing)
- Different from ML (and most other functional languages) in that it is PURELY functional
- (e.g., no variables, no assignment statements, and no side effects of any kind)

Most Important Features

- Uses lazy evaluation
- Has “list comprehensions,” which allow it to deal with infinite lists

HASKELL(cont..)

Examples

Fibonacci numbers (illustrates function definitions with different parameter forms)

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

2.Lazy evaluation

Infinite lists

e.g., positives = [0..]

squares = [n * n | n ¨ [0..]]

(only compute those that are necessary)

Applications of Functional Languages

APL is used for throw-away programs.

LISP is used for artificial intelligence

- Knowledge representation
- Machine learning
- Natural language processing
- Modeling of speech and vision

Scheme is used to teach introductory programming at a significant number of universities.

Comparing Functional and Imperative Languages

Imperative Languages:

Efficient execution

Complex semantics

Complex syntax

Concurrency is programmer designed

Functional Languages:

Simple semantics

Simple syntax

Inefficient execution

Programs can automatically be made concurrent

Scripting languages

Pragmatics

Scripting is a paradigm characterized by:

-use of scripts to glue subsystems together;

-rapid development and evolution of scripts;

-modest efficiency requirements;

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

Scripting languages(cont.)

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

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

Python

PYTHON was designed in the early 1990s by Guido van Rossum.

PYTHON borrows ideas from languages as diverse as PERL ,HASKELL ,and the object-oriented languages, skillfully integrating these ideas into a coherent whole.

PYTHON scripts are concise but readable, and highly expressive.

Values and types

PYTHON has a limited repertoire of primitive types: integer, real, and complex Numbers.

It has no specific character type; single-character strings are used instead.

its boolean values (named False and True) are just small integers.

PYTHON has a rich repertoire of composite types: tuples, strings, lists, dictionaries, and objects.

Variables, storage, and control

PYTHON supports global and local variables.

Variables are not explicitly declared,simply initialized by assignment.

PYTHON adopts reference semantics. This is especially significant for mutable values, which can be selectively updated.

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

PYTHON’s repertoire of commands include assignments, procedure calls, con-ditional (if- but not case-) commands, iterative (while- and for-) commands, and exception-handling commands.

PYTHON if- and while-commands are conventional.

Pythons reserved words

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;

Strongly typed python language:

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

Perl(weak)

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

Python(strong)

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

Bindings and scope

A PYTHON program consists of a number of modules, which may be grouped into packages.

Within a module we may initialize variables, define procedures, and declare classes

Within a procedure we may initialize local variables and define local procedures.

Within a class we may initialize variable components and define procedures (methods).

PYTHON was originally a dynamically-scoped language, but it is now statically scoped.

Binding and scope

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

```
S = 1
Def myfunc(x,y);
Z = 0

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

Procedural abstraction

PYTHON supports function procedures and proper procedures.

The only difference is that a function procedure returns a value, while a proper procedure returns nothing.

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

Python procedure

Eg :Def gcd (m, n):

```
p,q=m,n
while p%q!=0:
p,q=q,p%q
return q
```

Python procedure with Dynamic Typing

Eg: def minimax (vals):

```
min = max = vals[0]
for val in vals:
if val < min:
min = val
elif val > max:
max = val
return min, max
```

Data Abstraction

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

Modules and classes support encapsulation, using a naming convention to distinguish between public and private components.

A Package is simply a group of modules

A Module is a group of components that may be variables, procedures, and classes

Data abstraction(cont..)

A Class is a group of components that may be class variables, class methods ,and instance methods.

A procedure defined in a class declaration acts as an instance method if its first formal parameter is named self and refers to an object of the class being declared. Otherwise the procedure acts as a class method.

Data abstraction(cont..)

To achieve the effect of a constructor, we usually equip each class with an initialization method named ‘‘_init_’’; this method is automatically called when an object of the class is constructed.

PYTHON supports multiple inheritance: a class may designate any number of superclasses.

Separate Compilation

PYTHON modules are compiled separately.

Each module must explicitly import every other module on which it depends

Each module's source code is stored in a text file. Eg: program.py

When that module is first imported, it is compiled and its object code is stored in a file named program.pyc

Separate Compilation(cont..)

Compilation is completely automatic

The PYTHON compiler does not reject code that refers to undeclared identifiers.Such code simply fails if and when it is executed

The compiler will not reject code that might fail with a type error,nor even code that will certainly fail, such as:

```
def fail (x):
print x+1, x[0]
```

Module Library

PYTHON is equipped with a very rich module library, which supports string handling ,markup , mathematics, cryptography, multimedia, GUIs, operating system services ,internet services, compilation, and so on.

Unlike older scripting languages, PYTHON does not have built-in high-level string processing or GUI support , so module library provides it.