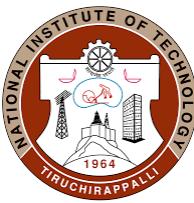


CSPC31: Principles of Programming Languages

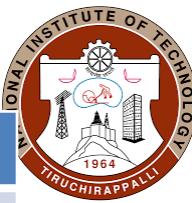
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Programming Languages

Sl. No.	Programming Language	Website
1.	C, C++, Fortran and Ada	gcc.gnu.org
2.	C#	microsoft.com
3.	Java	java.sun.com
4.	Haskell	haskell.org
5.	Scheme	www.plt-scheme.org/software/drscheme
6.	Perl	www.perl.com
7.	Python	www.python.org
8.	Ruby	www.ruby-lang.org/en/



Chapter 1 - Preliminaries



Objectives

- Reasons for studying
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-offs
- Implementation Methods



Reasons for Studying

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



Programming Domains

Sl. No.	Domain	Requirement	Programming Language
1.	Scientific Applications	<ul style="list-style-type: none">• Floating-point arithmetic computations• Arrays and Matrices• Loops and Selections	Fortran
2.	Business Applications	<ul style="list-style-type: none">• Produce detailed reports• Describe and store decimal numbers and character data• Ability to specify decimal arithmetic operations	COBOL
3.	Artificial Intelligence	<ul style="list-style-type: none">• Use of symbolic rather than numeric computations• Linked List of data	LISP
4.	Systems Programming	<ul style="list-style-type: none">• Fast execution• Have low-level features that allow the software interface to external device to be written	PL/S, BLISS, Extended ALGOL UNIX OS – C
5.	Web Software	<ul style="list-style-type: none">• Dynamic web content• Request execution of a separate program on the Web Server to provide dynamic content	XHTML, Java, JS, PHP



Language Evaluation Criteria

- Readability
- Writability
- Reliability
- Cost



Language Evaluation Criteria

Table 1.1 Language evaluation criteria and the characteristics that affect them

Characteristic	CRITERIA		
	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•



Readability

- Programming languages were constructed from the p-o-v of the computer
- Maintenance
- Shift from a focus on machine orientation to a focus on human orientation



Readability

- Overall Simplicity
- Orthogonality
- Control Statements
- Data Types and Structures
- Syntax Design



Overall Simplicity - Miscellaneous

- Keep the language as simple as possible
- Eg: Dictionary has lot of words
 - Are we learning them all?
 - Are we using them all in our day-to-day life?
 - Are we starting to speak only after learning all those words?
- Same for programming language



Overall Simplicity

- Language with more basic constructs are difficult to learn
- Feature Multiplicity

count = count + 1

count += 1

count++

++count

- Operator Overloading

Demerits: Too much simplicity also raises concern –
Assembly Language

Orthogonality - Miscellaneous

- Combinations
- Keep the rule as simple as possible while producing combinations
- Eg: I, am, eating, food

Combination	Correctness
I am eating food	✓
Am I eating food	✓
I eating am food	✗
Food I eating am	✗
.....	✗
.....	✗

Eg: int, char

Expression: $c = a + b$

a	b	Correctness
int	int	✓
char	char	✓
int	char	✗
char	int	✗



Orthogonality

- Ability to combine a relatively small set of primitive constructs in a relatively small number of ways to build DS
- Eg: 4 primitive data types (int, float, double and character); 2 type operators (array and pointer)
- Task:** Add 2 nos. that reside in either memory or registers and replace one of the two values with the sum

Operand 1	Operand 2	Result
Reg	Reg	Reg
Reg	Mem	Reg
Mem	Reg	Mem
Mem	Mem	Mem



Orthogonality

- IBM mainframe computers vs VAX series of mini-computers
- IBM

A Reg1, memory_cell
AR Reg1, Reg2

Rules:

1. At least one variable should reside in reg
2. Result can only be stored in a reg
3. Use different codes – A & AR

Operand 1	Operand 2	Result	Correctness	Code
Reg	Reg	Reg	✓	AR
Reg	Mem	Reg	✓	A
Mem	Reg	Mem	✗	-
Mem	Mem	Mem	✗	-

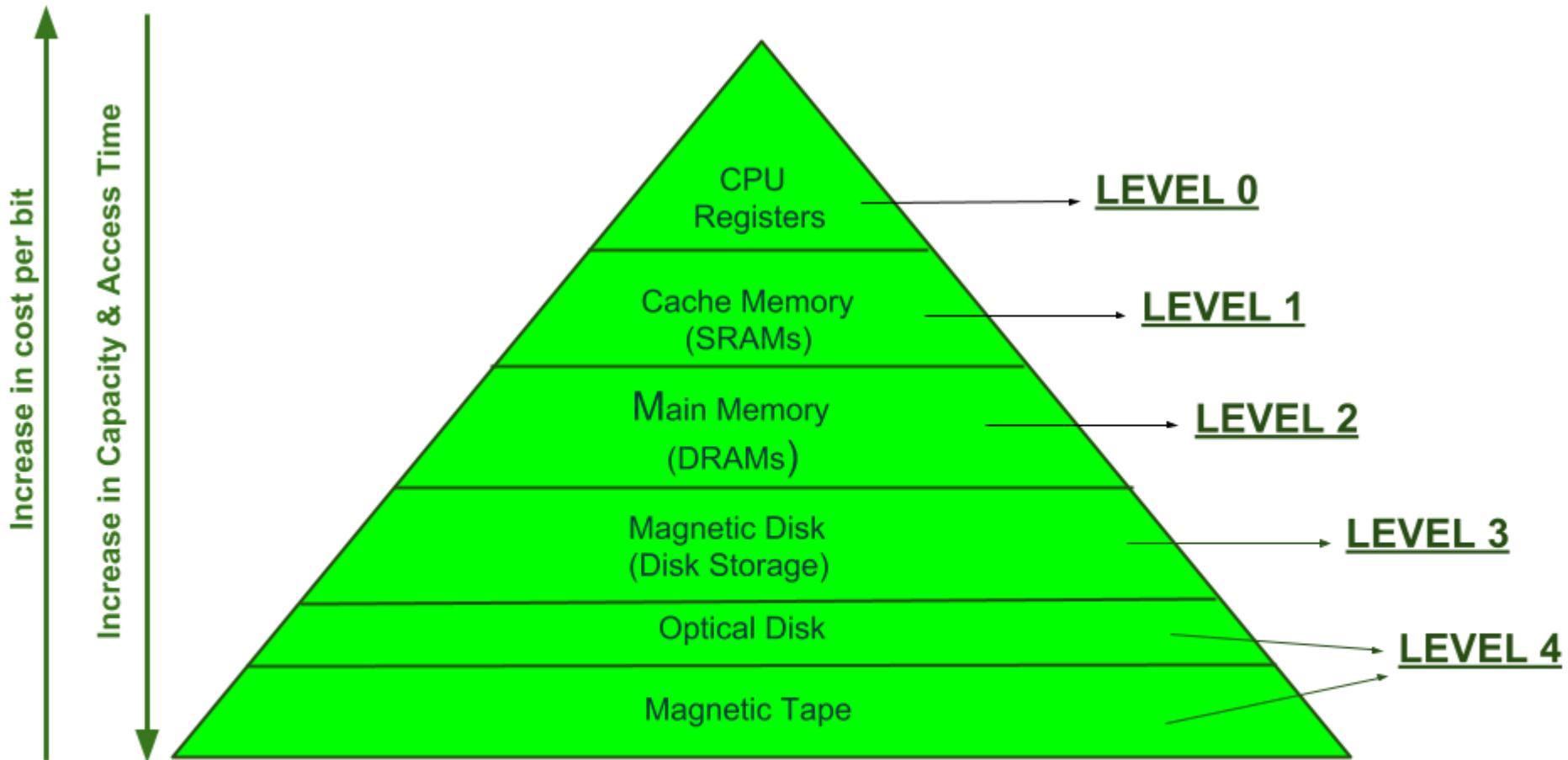
- VAX

ADDL operand_1, **operand_2**

No Rules

Operand can be a register or a memory cell → VAX instruction design is orthogonal

Memory Hierarchy





Orthogonality

- Issues in C Programming (Array and Struct)

Sl. No.	Struct	Array
1.	Records can be returned from functions	Arrays cannot
2.	Member of a structure can be any data type except void or a structure of the same type	An array element can be any data type except void or a function
3.	Parameters are passed by value	Parameters are passed by reference

- Context dependence: $a + b$ [$a \rightarrow \text{float}$; $b \rightarrow \text{int}$]
- Eg for Orthogonality: [Algol 68](#)
- **Demerit:** Too much orthogonality can also cause problem



Control Statement

- Decision Making Statement (if – else)
- Selection Statement (switch – case)
- Iteration Statement (while, do – while, for)
- Jump Statement (goto)



Control Statement

- In Fortran:

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

- In C:

```
while (incr < 20)  
{  
  while (sum <= 100)  
  {  
    sum += incr;  
  }  
  incr++;  
}  
}
```



Data Types and Structures

- Data Type

timeOut = 1

timeOut = **true** ← More Meaningful

- Record

Sl. No.	Name	Age	Employee_Number	Salary
1.	Bala	25	100	10000
2.	Krishnan	30	110	20000
.....
.....
100	Krish	35	120	30000



Data Types and Structures

- In Fortran:
 - Employee Records must be stored in the form of arrays
 - Character (Len = 30) :: Name (100)
 - Integer:: Age (100), Employee_Number (100)
 - Real:: Salary (100)
 - Name₁, Age₁, Employee_Number₁, Salary₁



Data Types and Structures

In C:

```
struct employee_record  
{  
    char *name;  
    int age;  
    int employee_number;  
    int salary;  
};
```

```
int main()  
{  
    struct employee_record one;  
    one.name = "bala";  
    one.age = 10;  
    one.employee_number = 100;  
    one.salary = 10000;  
    printf("%d", one.age);  
}
```



Syntax Design

- Identifier Forms
 - ✓ Fortran 77 - At most 6 characters
 - ✓ ANSI Basic - 1 letter or 1 letter followed by a digit
- Special Words (while, class and for)
 - ✓ C - { } ← Not clear which loop ends
 - ✓ Ada - end if (terminates selection construct), end loop (terminates a loop construct)
- Usage of Special Words as names for program variables
 - ✓ Ada - "Do" and "End" are legal variable names ← Confuse
- Form and Meaning
 - ✓ Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability
 - ✓ C Programming – Static keyword
 - If used on the definition of a variable inside a function, it means the variable is created at compile time
 - If used on the definition of a variable that is outside all functions, it means the variable is visible only in the file in which its definition appears; that is, it is not exported from that file
- grep (g/regular_expression/p) → Prints all lines in a file that contain substrings that match the regular expression

$$26^1 + 26^2 + 26^3 + 26^4 + 26^5 + 26^6$$

```
int c;  
int a = 0;
```



Summary - Readability

Sl. No.	Characteristic	Conclusion	Demerits
1.	Overall Simplicity	Keep the programming language as simple as possible	Too much simplicity is also a problem
2.	Orthogonality	Keep the combination rules simple and try to accept all possible combinations	Too much orthogonality is also a problem
3.	Control Statements	Must have enough control statements (while, for, etc.)	-
4.	Data Types and Structures	Must have enough Data Types and Structures	-
5.	Syntax Design	Size of Identifier, Special Words, Form and Meaning	-



Writability

- How easy a language is to create programs
- Characteristics that affect readability also affects writability
- Fortran is ideal for creating 2D arrays
- COBOL is ideal for producing financial reports with complex formats



Writability

- Simplicity and Orthogonality
- Support for Abstraction
- Expressivity



Simplicity and Orthogonality

- Large number of different constructs can raise concern
 - Misuse of some features and a disuse of others
- Have a smaller number of primitive constructs and a consistent set of rules for combining them
 - Quickly design a solution after learning only a simple set of primitive constructs
- **Demerit:** Too much orthogonality can be detriment to writability
 - Errors in programs can go undetected when nearly any combination of primitives is legal
 - **Eg:** $a + b$ [$a \rightarrow \text{float}$; $b \rightarrow \text{int}$]



Support for Abstraction

- Ability to define and use complicated operations ignoring minor details
- Two categories
 - ✓ Process
 - ✓ Data



Process Abstraction

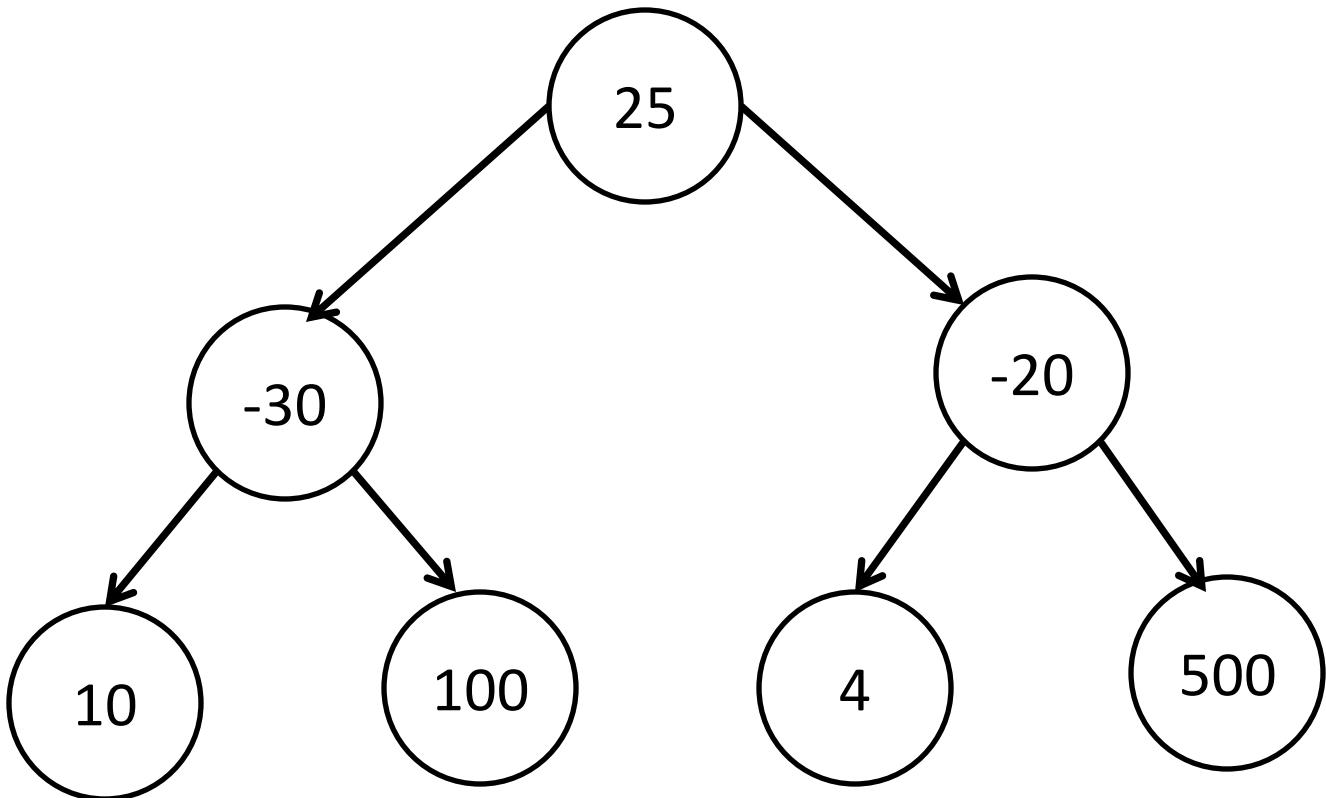
- Use of subprogram to achieve a task
- Sorting algorithm
 - ✓ Insertion Sort
 - ✓ Merge Sort
 - ✓ Quick Sort



Data Abstraction

- Binary tree storing integer data in its nodes
- Each node will have:
 - ✓ An integer value
 - ✓ A left pointer pointing to another node
 - ✓ A right pointer pointing to another node

Binary Tree





Data Abstraction

- Fortran:
 - Does not support pointers and dynamic storage management with a heap
 - Soln: Three parallel integer arrays
- C++ and Java:
 - Use an abstraction of a tree node in the form of a simple class with two pointers and an integer



Expressivity

- Convenient rather than cumbersome
- Increment operation is more convenient by using “++” operator
- “For” is better than “while” for writing “counting loops”



Expressivity

- While Loop in Java:

```
int count = 1;  
while (count <= 10)  
{  
    out.println(count);  
    count = count + 1;  
}
```

- For Loop in Java:

```
for (int count = 1; count <= 100; count++)  
{  
    out.println(count);  
}
```



Summary - Writability

Sl. No.	Characteristic	Remarks
1.	Simplicity and Orthogonality	Too much orthogonality is a concern
2.	Support for Abstraction	<ul style="list-style-type: none">• Process Abstraction• Data Abstraction
3.	Expressivity	<ul style="list-style-type: none">• Convenience matters• Loops



Reliability

- Works as per specifications under all conditions
- Evaluation Features
 - ✓ Type Checking
 - ✓ Exception Handling
 - ✓ Aliasing
 - ✓ Readability and Writability



Type Checking

- Testing for Type errors
 - Compiler
 - During Program Execution
- The earlier the errors are detected, it is less expensive to make the repairs
- Compile-time type checking is more desirable
- Run-time checking is expensive
- Java programs do type checking of nearly all variables and expressions at compile-time
- Earlier, C language does not perform type checking
 - Type of actual and formal parameters mismatch
 - Unix – Utility program named “lint” → checks C programs for such problems

```
int a = 0, b = 0;  
int c = add(a, b);  
int add(float d, int e)
```



Compile-Time vs Run-Time

Compile Time:

```
#include<stdio.h>
public class CompileDemo
{
    void main()
    {
        int x = 100;
        int y = 155;
        // semicolon missed
        printf("%d", (x, y))
    }
}
```

O/P: error: expected ';' before
'}' token

Run Time:

```
#include<stdio.h>
public class RuntimeDemo
{
    void main()
    {
        int n = 9;
        div = 0;
        div = n/0;
        printf("result = %d", div);
    }
}
```

O/P: warning: division by zero [-Wdiv-by-zero] div = n/0;



Binary Representation

Integer 23:

000000000000000000000000000010111

Float 23.0:

010000011011000000000000000000000



Type Checking - Example

```
#include <stdio.h>
float add(float num1, float num2)
{
    return num1 + num2;
}
int main()
{
    int a = 5, b = 10;
    float c = add(a, b);
    printf("%f", c);
    return 0;
}
O/P: 15.000000
```



Exception Handling

- Ability to intercept run-time errors, take corrective actions and then continue
- Ada, C++, Java – Extensive capabilities for exception handling
- C, Fortran – Do not have much facility



Exception Handling

```
public class MyClass
{
    public static void main(String[ ] args)
    {
        try
        {
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]);
        }
        catch (Exception e)
        {
            System.out.println("Something went wrong.");
        }
    }
}
```

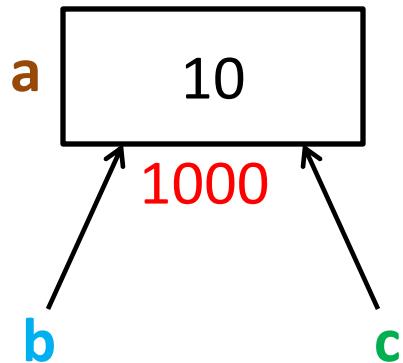
O/P: Something went wrong.



Aliasing

- Two or more distinct names that can be used to access the same memory cell
- Dangerous feature in a programming language
- Allow some kind of aliasing – Two pointers set to point to the same variable
- Changing the value pointed to by one of the two changes the value referenced by the other
- Used to overcome deficiencies in the language's data abstraction facilities
- Many languages greatly restrict aliasing to increase their reliability

Aliasing



```
void main()
{
    int a = 10;
    int *b = &a;
    int *c = &a;
    printf("b = %d\n", *b);
    printf("c = %d\n", *c);
    *b = 15;
    printf("b = %d\n", *b);
    printf("c = %d\n", *c);
}
```

O/P: b = 10 c = 10 b = 15 c = 15



Aliasing

In C:

```
int gfg(int* a, int* b)
{
    *b = *b + 10;
    return *a;
}
```

```
int main()
{
    int data = 20;
    int result = gfg(&data, &data);
    printf("%d ", result);
}
```

O/P: 30



Readability and Writability

- Program written in a language that does not support natural ways to express the required algorithms will use unnatural approaches
- **Eg:** “go to” instead of “while” in Fortran
- Easier a program is to write, more likely it is to be correct
- Readability affects reliability in writing and maintenance phases
- Programs that are difficult to read will also be difficult to write and modify



Control Statement

- In Fortran:

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

- In C:

```
while (incr < 20)  
{  
  while (sum <= 100)  
  {  
    sum += incr;  
  }  
  incr++;  
}  
}
```



Reliability - Summary

Sl. No.	Characteristic	Remarks
1.	Type Checking	Run-time checking is expensive
2.	Exception Handling	<ul style="list-style-type: none">• Intercept run-time errors• Take corrective action• Continue
3.	Aliasing	<ul style="list-style-type: none">• Two or more distinct names that can be used to access the same memory cell• Avoid aliasing
4.	Readability and Writability	Unnatural approaches are less likely to be correct for all possible situations



Cost

- Cost of training programmers to use the language
 - Simplicity
 - Orthogonality
 - Experience
- Cost of writing programs in the language
 - Closeness in purpose to the particular application
- Cost of compiling programs in the language
 - Running first generation Ada compilers are costly
- Cost of the language implementation system
 - Java -> Free availability of compiler/interpreter systems
- Cost of poor reliability
 - Failure in critical system like NPP or an X-ray machine



Cost

- Cost of executing programs written in a language is greatly influenced by the language's design
 - Many run-time type checks will prohibit fast code execution
 - Trade-off can be made between compilation cost and execution speed of compiled code
 - Optimization -> Collection of techniques that compilers use to decrease the size and/or increase the execution speed
 - If little or no optimization is done, then compilation can be done much faster
 - The extra compilation effort results in faster code execution
 - Trade-off: 1. Laboratory environment; 2. Production environment
- Cost of Maintaining programs
 - Includes both corrections and modifications to add new capabilities



Cost - Summary

Sl. No.	Characteristic
1.	Training programmers to use the language
2.	Writing programs in the language
3.	Compiling programs in the language
4.	Influence by the language design
5.	Language implementation system
6.	Poor reliability
7.	Maintaining the programs



Other Evaluation Criteria

- Portability
 - Ease with which programs can be moved from one implementation to another
- Generality
 - Applicability to a wide range of applications
- Well-definedness
 - Completeness and precision of the language's official defining document

Note:

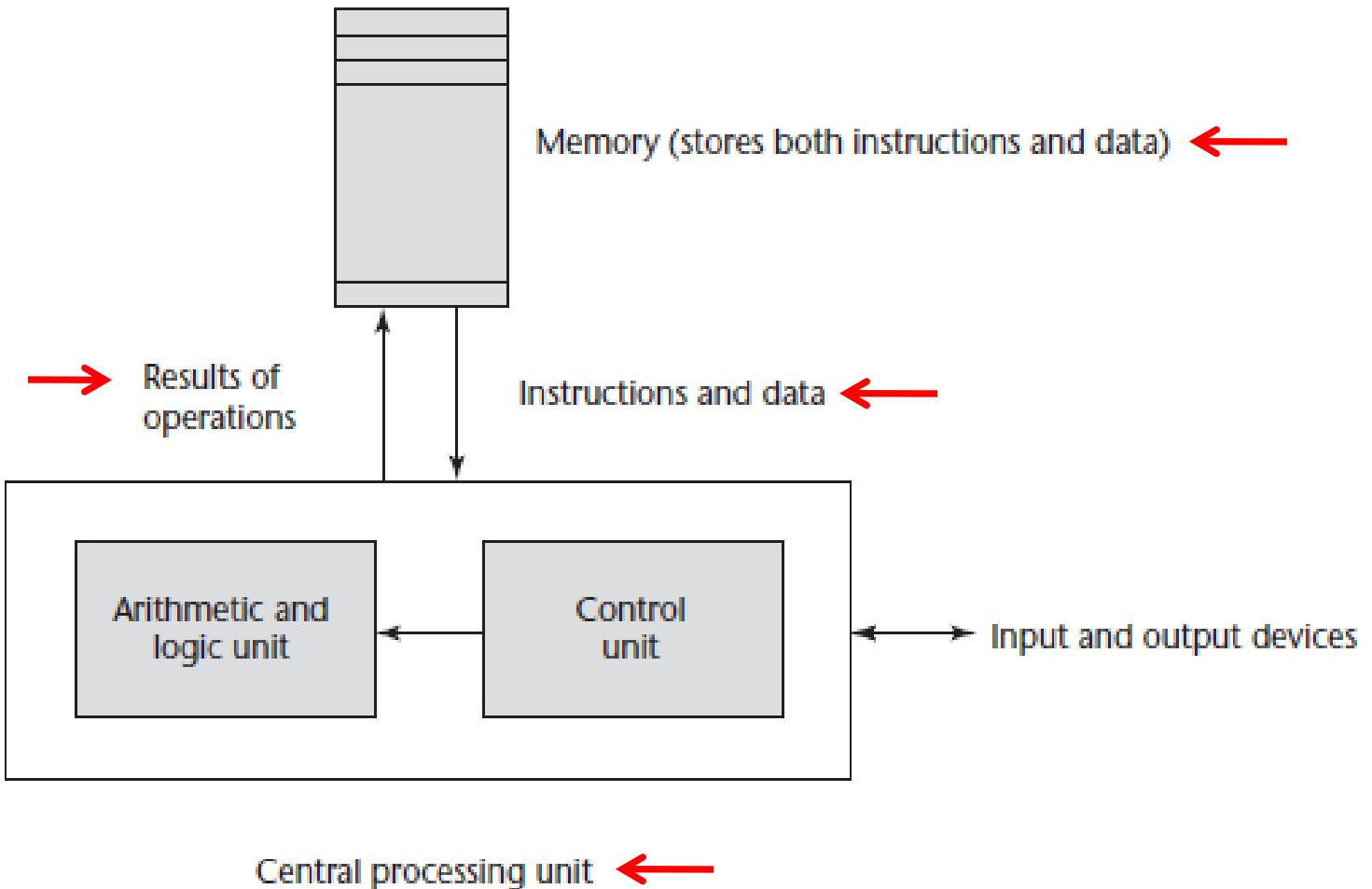
- Language implementers mainly focus on the difficulty of implementing the constructs and features
- Language users focus on writability first and readability later



Influences on Language Design

- Computer Architecture
- Programming Methodologies

Computer Architecture



Von Neumann Architecture



Computer Architecture

- Languages which follow this architecture is called Imperative Languages
- Central features of Imperative Languages
 - Variables → Memory cells
 - Assignment Statements → Piping operation
 - Iterative form of repetition -> Implement repetition
- Iteration is fast in Von Neumann
 - Repeating a section of code requires a simple branch instruction
 - Discourages the use of recursion for repetition
- Process of executing a machine code program is called Fetch-execute cycle
- Functional or applicative languages -> Applying functions to given parameters



Iteration vs Recursion

```
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```

initialize the PC
repeat forever
 Fetch the instruction pointed to by the PC
 Increment the PC to point at the next instruction
 Decode the instruction
 Execute the instruction
End **repeat**



Programming Methodologies

- Late 1960 and early 1970 brought intense analysis
 - Software development process
 - Programming language design
- Hardware cost ↓ and programmer cost ↑
- Top-down design and stepwise refinement
- Type checking and inadequacy of control statements
- Late 1970s, shift from procedure-oriented to data-oriented
 - Data design, focus on the use of ADT to solve problems
- First language -> SIMULA 67
- Most languages designed since late 1970s support data abstraction



Programming Methodologies

- Latest is object-oriented design
 - Data abstraction
 - Inheritance -> Reuse of existing software
 - Dynamic method binding -> Flexible use of inheritance
 - Smalltalk -> Supported object-oriented design
 - Imperative programming languages -> Ada 95, Java, C++
 - Functional programming -> CLOS
 - Logic programming -> Prolog++
- Procedure-oriented programming -> Concurrency
 - Need for language facilities for creating and controlling concurrent program units
 - Ada, Java, C#

Static vs Dynamic Method Binding



```
class Human
{
    public static void walk()
    {
        System.out.println("Human
                           walks");
    }
}

public class Boy extends Human
{
    public static void walk()
    {
        System.out.println("Boy
                           walks");
    }
}
```

static, private or final method in a class
-> Compile time

```
public static void main( String
                       args[])
{
    /* Reference is of Human type
       and object is * Boy type */
    Human obj = new Boy(); ←
```

```
/* Reference is of Human type
   and object is * of Human type
*/
```

```
Human obj2 = new Human();
```

```
obj.walk();
obj2.walk();
}
```

O/P: **Human walks**
Human walks



Language Categories

- Imperative (Expressions and Assignment Statements)
- Functional
- Logic -> Prolog
- Object-Oriented
- Scripting Languages -> Perl, JS, Ruby
- Visual Language
 - Visual Basic -> VB .NET
 - Drag-and-drop
 - 4th Gen Languages



Language Categories

- Markup / Programming Hybrid Languages -> XHTML
 - Used to specify the layout of information in Web Documents
 - Java Server Pages Standard Tag Library (JSTL) and eXtensible Stylesheet Language Transformation (XSLT)
- Special-Purpose Languages
 - Report Program Generator (RPG) -> Business Reports
 - Automatically Programmed Tools (APT) -> Program numerically-controlled machine tools
 - General Purpose Simulation System (GPSS) -> Systems Simulation



Language Design Trade-offs

- Two conflicting criteria
 - ✓ Reliability
 - ✓ Cost of execution
- Eg:
 - ✓ Java -> All references to array elements will be checked
 - ✓ C does not do this checking
- C executes faster whereas Java is reliable
- Java -> Traded execution efficiency for reliability



Language Design Trade-offs

- APL Programming Language -> Powerful set of operators for array operands
 $a[1] + a[2] - a[3]$, where “a” is an array
- New symbols had to be included in APL to represent the operators
- Many APL operators can be used in a single long, complex expression
- Adv:
 - High degree of expressivity
 - Application involving many array operations -> APL is preferred
 - Huge amount of computation can be specified in a very compact program
 - Poor readability
- Daniel McCracken -> Took 4 hours to read and understand a four-line APL program
- APL -> Traded Readability for Writability



Language Design Trade-offs

- Pointers of C++ can be manipulated in a variety of ways leading to highly flexible addressing of data

p -> Pointer; *p *p++ * ++p ++*p
- **Problem:**
 - Referencing NULL pointer -> int *p = 0;
 - Deleting twice -> delete p; delete p;
- Reliability problem -> Pointers are not included in Java
- Task of choosing constructs and features when designing a programming language requires many compromises and trade-offs



Miscellaneous

- In C programming language, *p represents the value stored in a pointer. ++ is increment operator used in prefix and postfix expressions. * is dereference operator.
- Precedence of prefix ++ and * is same and both are **right to left associative**.
- Precedence of postfix ++ is higher than both prefix ++ and * and is **left to right associative**.

```
#include <stdio.h>
int main()
{
    int arr[] = {20, 30, 40};
    int *p = arr;
    int q;
    //value of p (20) incremented by 1
    //and returned
    q = ++*p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d, q = %d \n", arr[0], arr[1], *p, q);
    //value of p (20) is returned
    //pointer incremented by 1
    q = *p++;
    printf("arr[0] = %d, arr[1] = %d, *p = %d, q = %d \n", arr[0], arr[1], *p, q);
    //pointer incremented by 1
    //value returned
    q = *++p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d, q = %d \n", arr[0], arr[1], *p, q);
    return 0;
}
```

O/P:

arr[0] = 21, arr[1] = 30, *p = 21, q = 21
arr[0] = 21, arr[1] = 30, *p = 30, q = 21
arr[0] = 21, arr[1] = 30, *p = 40, q = 40

<https://www.tutorialspoint.com/difference-between-plusplus-p-pplusplus-and-plusplusp-in-c>



Implementation Methods

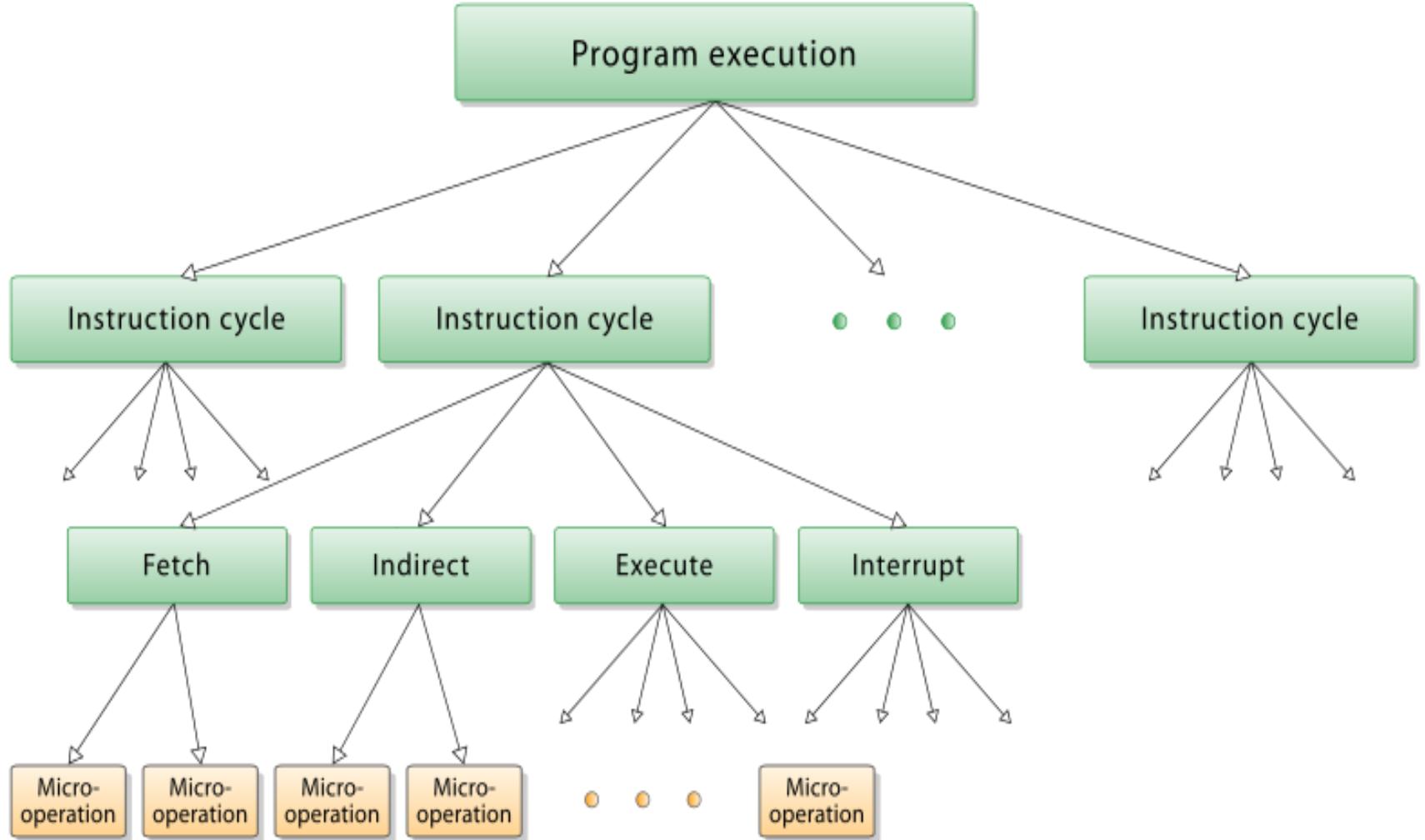
- Two primary components – Internal Memory and Processor
- Internal Memory -> Store Programs and Data
- Processor -> Provides a realization of a set of primitive operations or machine instructions such as Arithmetic and Logic Operations
- Machine instructions -> Macroinstructions
- Macroinstructions are implemented with a set of instructions called microinstructions



Microinstruction

- Performs basic operations on data stored in one or more registers
 - ✓ Includes transferring data between registers or between registers and external buses of the CPU
 - ✓ Performs arithmetic or logical operations on registers
- In a typical fetch-decode-execute cycle, each step of a macro-instruction is decomposed during its execution so the CPU determines and steps through a series of micro-operations

Microinstruction





Instruction Cycle (Fetch-decode-execute cycle)

- Main job of the CPU is to execute programs using the fetch-decode-execute cycle (also known as the instruction cycle) -> Cycle begins as soon as you turn on a computer
- To execute a program, the program code is copied from secondary storage into the main memory
- CPU's program counter is set to the memory location where the first instruction in the program has been stored, and execution begins -> The program is now running
- In a program, each machine code instruction takes up a slot in the main memory. These slots (or memory locations) each have a unique memory address. The program counter stores the address of each instruction and tells the CPU in what order they should be carried out
- When a program is being executed, the CPU performs the fetch-decode-execute cycle, which repeats over and over again until reaching the STOP instruction



Instruction Cycle (Fetch-decode-execute cycle)

Summary of the fetch-decode-execute cycle

1. The processor checks the program counter to see which instruction to run next.
2. The program counter gives an address value in the memory of where the next instruction is.
3. The processor fetches the instruction value from this memory location.
4. Once the instruction has been fetched, it needs to be decoded and executed. For example, this could involve taking one value, putting it into the **ALU**, then taking a different value from a **register** and adding the two together.
5. Once this is complete, the processor goes back to the program counter to find the next instruction.
6. This cycle is repeated until the program ends.

<https://www.bbc.co.uk/bitesize/guides/z2342hv/revision/5#:~:text=The%20main%20job%20of%20the,storage%20into%20the%20main%20memory.>



Implementation Methods

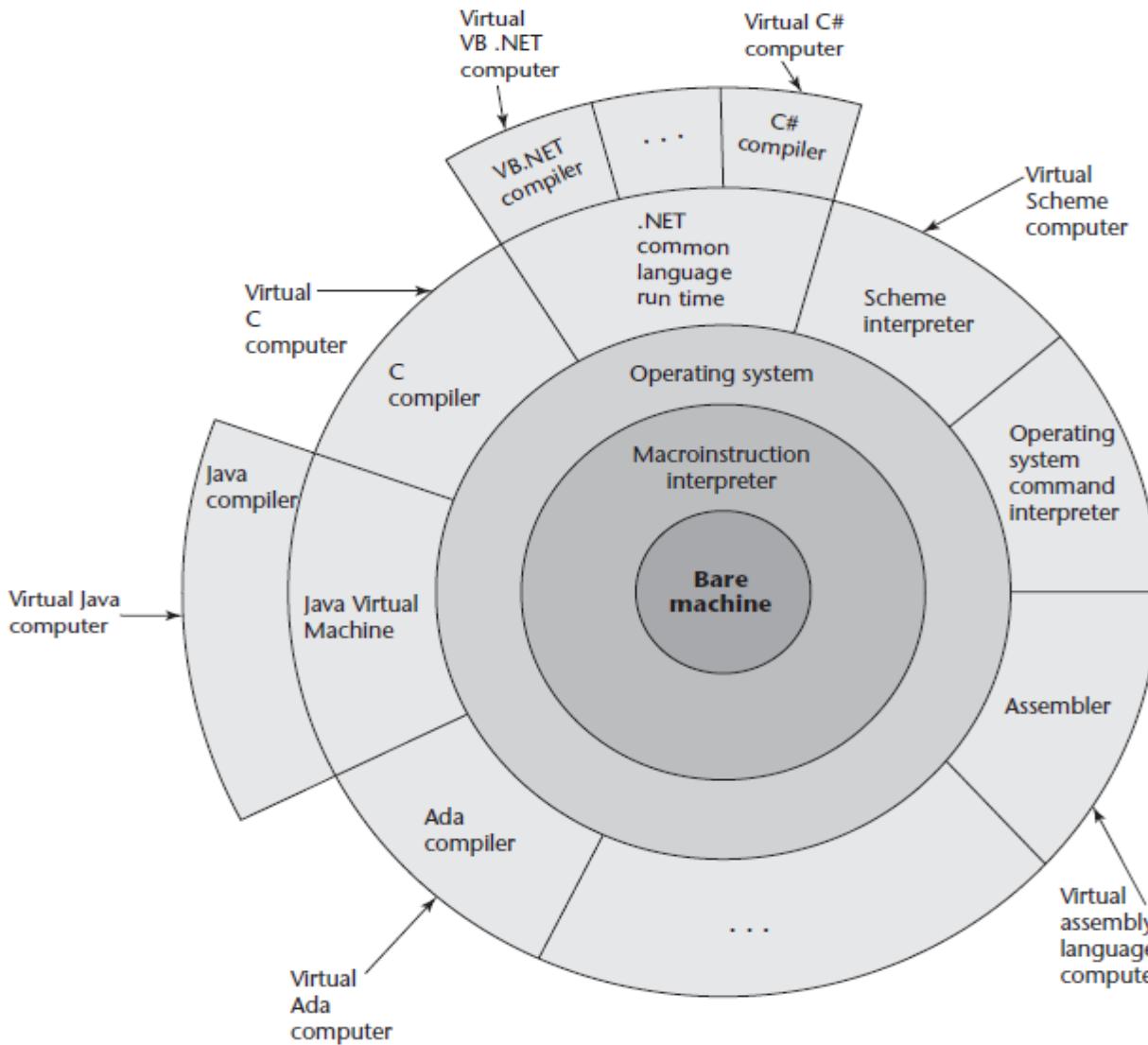
- Machine language of the computer is its set of instructions
- Its own machine language is the only language that most hardware computers “understand”
- Computer could be designed and built with a particular high-level language as its machine language
 - ✓ Complex
 - ✓ Expensive
 - ✓ Highly inflexible
- Practical machine design choice -> A very low-level language (provides the most commonly needed primitive operations) + an interface to programs in higher-level languages



Implementation Methods

- Language implementation system is not the only software on the computer
- Large collection of programs called Operating system
 - ✓ System resource management
 - ✓ Input and Output Operations
 - ✓ File Management System
 - ✓ Text and/or Program Editors
- Language implementation systems need many of the OS facilities
- Interface to the OS rather than directly to the processor (in machine language)

Implementation Methods





Implementation Methods

- Layers can be thought of as virtual computers, providing interfaces to the user at higher levels
- Eg: An OS and C Compiler provide a virtual C Computer
- With other compilers, a machine can become other kinds of virtual computers
- Most computer systems provide several different virtual computers
- User programs form another layer over the top of the layer of virtual computers

Implementation of Programming Languages



- Compiler
- Pure Interpretation
- Hybrid Implementation Systems
- Preprocessors

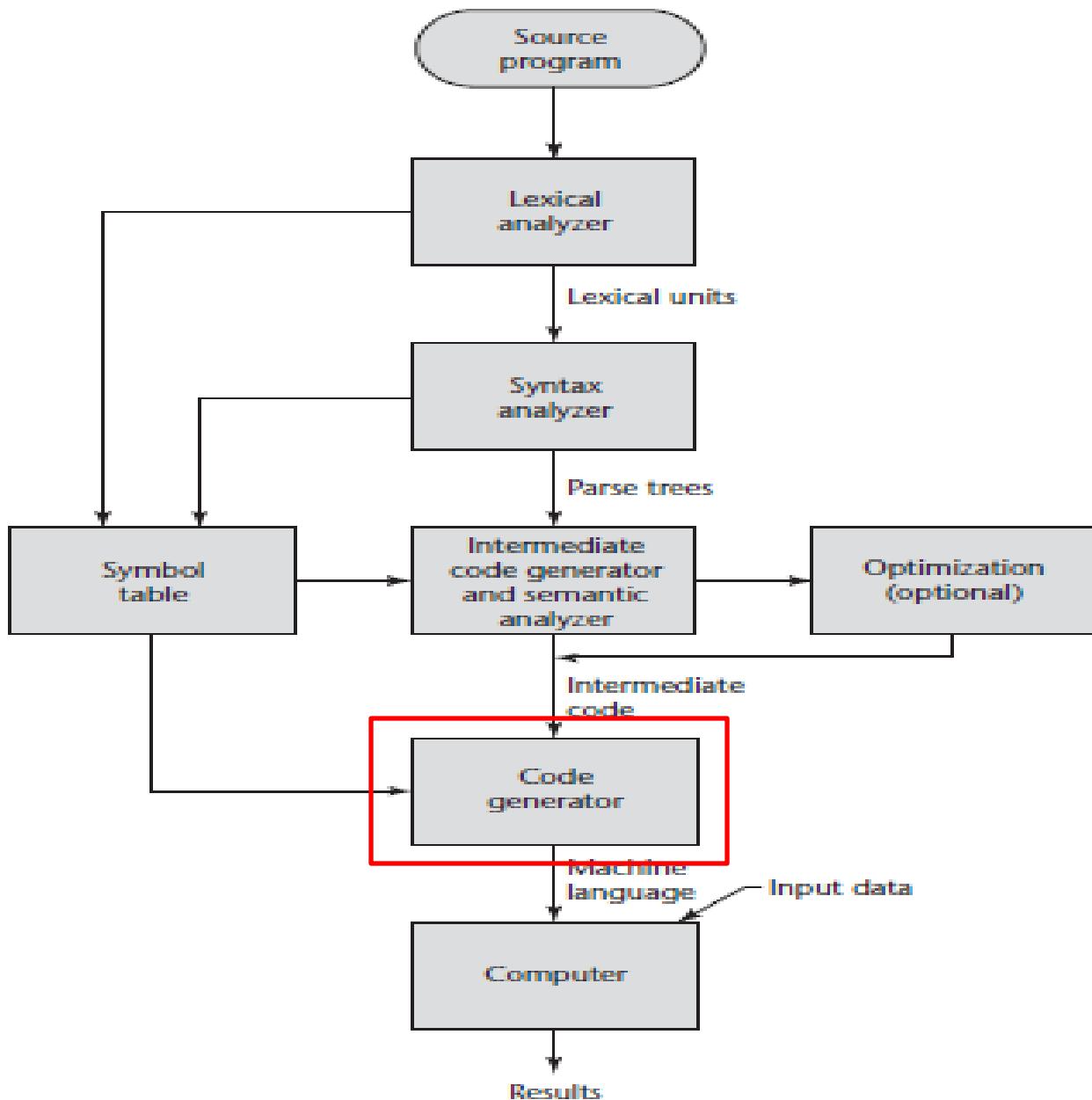


Compilation

- Programs can be converted into machine language
 - ✓ Compiler Implementation
 - ✓ Fast execution
 - ✓ C, COBOL and Ada
- Language that a compiler translates -> Source Language
- Process of compilation and program execution takes place in several phases
 - ✓ Lexical Analyzer
 - ✓ Syntax Analyzer
 - ✓ Intermediate Code Generator
 - ✓ Code Optimization
 - ✓ Code Generation

<https://blog.cloudflare.com/how-to-execute-an-object-file-part-1/>

Compilation





Lexical Analyzer

- Gathers the characters of the source program into Lexical units
 - ✓ Identifiers
 - ✓ Special words
 - ✓ Operators
 - ✓ Punctuation Symbols
- Ignores comments



Lexical Analyzer

Portion = Initial + Rate * 60

Sl. No.	Token Value	Token Type
1.	Portion	Identifier
2.	=	Assignment Operator
3.	Initial	Identifier
4.	+	Arithmetic Addition Operator
5.	Rate	Identifier
6.	*	Arithmetic Multiplication Operator
7.	60	Constant

$\text{id}_1 = \text{id}_2 + \text{id}_3 * 60$

<https://www.guru99.com/compiler-design-lexical-analysis.html>



Symbol Table

- Serves as a DB for compilation process
- Contents -> Type and attribute information of each user-defined name
- Placed by the lexical and syntax analyzers
- Used by the semantic analyzer and Code Generator

$$\text{Portion} = \text{Initial} + \text{Rate} * 60$$

Variable Name	Data Type	Mapped Identifier _#
Portion	int	id_1
Initial	int	id_2
Rate	int	id_3

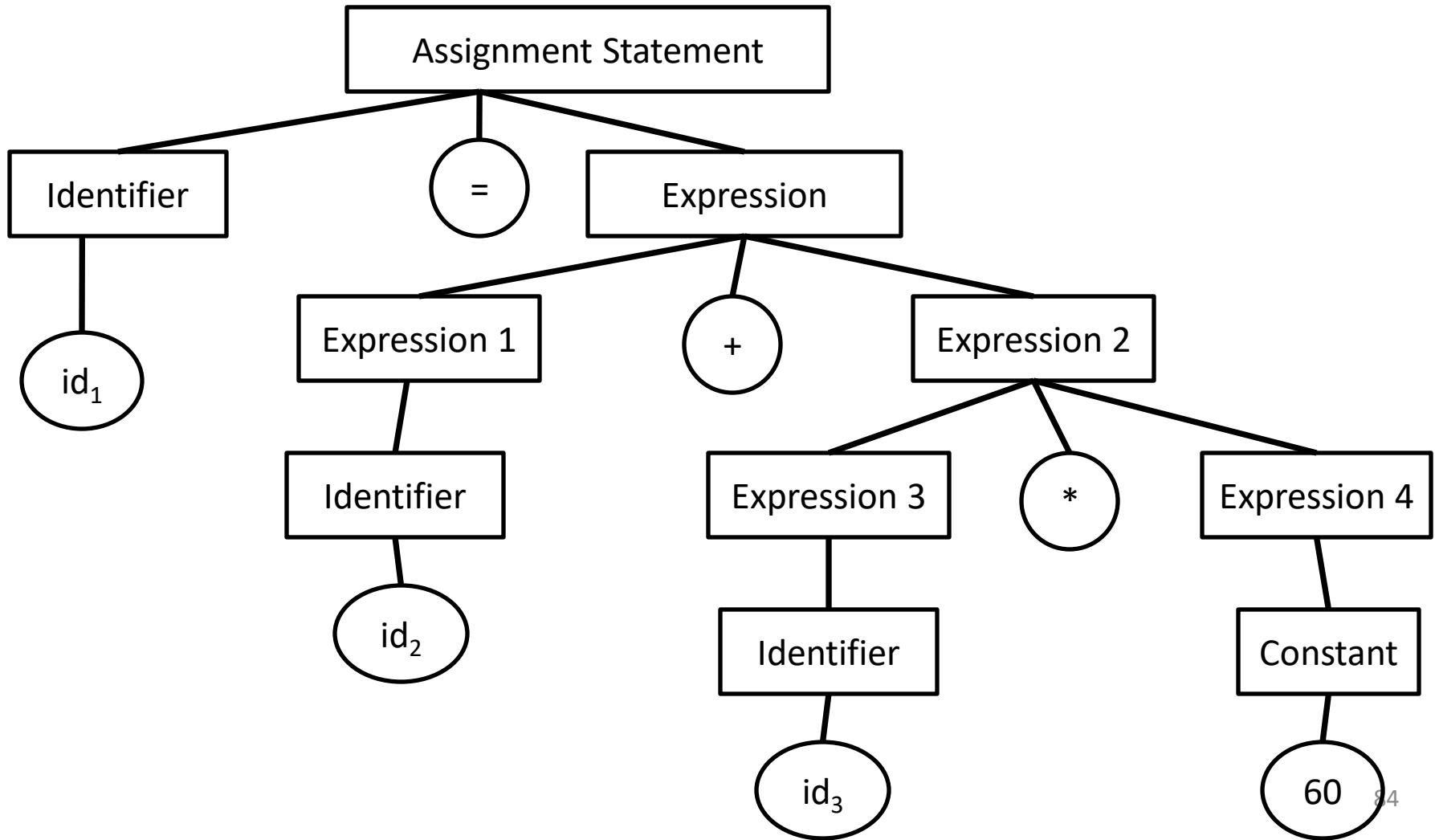


Syntax Analyzer

- Takes the lexical units and uses them to construct parse trees
- Represents the syntactic structure of the program
- No actual parse tree structure is generated
- Information that would be required to build a tree is generated and used directly

Syntax Analyzer

$\text{id}_1 = \text{id}_2 + \text{id}_3 * 60$





Intermediate Code Generator

- Produces a program at an intermediate level
- Looks very much like assembly language
- Semantic analyzer is an integral part
 - Checks for errors -> Typos

$$\text{id}_1 = \text{id}_2 + \text{id}_3 * 60$$

$\text{temp}_1 = \text{id}_3 * 60$

$\text{temp}_2 = \text{id}_2 + \text{temp}_1$

$\text{id}_1 = \text{temp}_2$

General Format:

identifier = operand₁ operator operand₂
or

identifier = unary-operator operand



Code Optimization

- Improves programs -> smaller or faster or both
- Some compilers are incapable of doing any significant optimization
 - Used in situations where execution speed is less important than compilation speed
 - Eg: Laboratory for beginners
- Commercial and industrial situations -> Optimization is routinely desirable
- Optimization is done on intermediate code



Code Optimization

$$\mathbf{id}_1 = \mathbf{id}_2 + \mathbf{id}_3 * 60$$

$\text{temp}_1 = \mathbf{id}_3 * 60$

$\text{temp}_2 = \mathbf{id}_2 + \text{temp}_1$

$\mathbf{id}_1 = \text{temp}_2$

$\text{temp}_1 = \mathbf{id}_3 * 60$

$\mathbf{id}_1 = \mathbf{id}_2 + \text{temp}_1$



Code Generator

- Translates the optimized intermediate code into an equivalent machine language program

$$\mathbf{id}_1 = \mathbf{id}_2 + \mathbf{id}_3 * 60$$

$\text{temp}_1 = \mathbf{id}_3 * 60$

$\mathbf{id}_1 = \mathbf{id}_2 + \text{temp}_1$

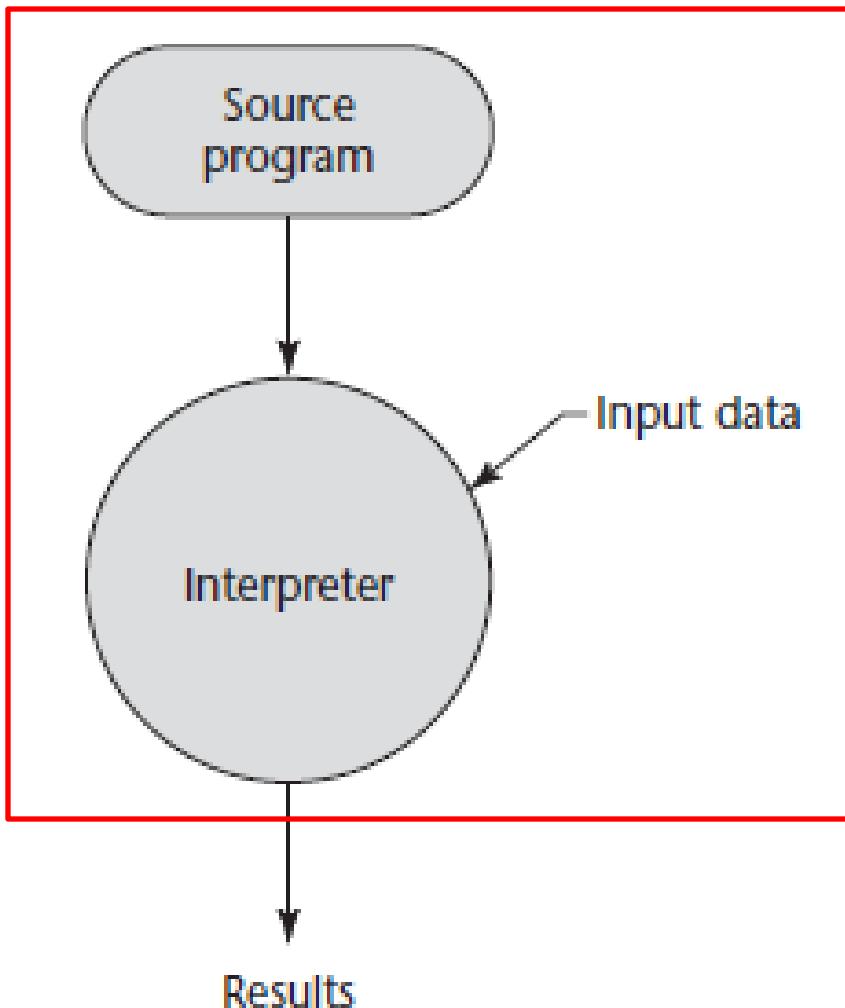
MOV \mathbf{id}_3 , R₁
MUL #60, R₁
MOV \mathbf{id}_2 , R₂
ADD R₁, R₂
MOV R₂, \mathbf{id}_1



Discussion

- Machine language generated by a compiler can be executed directly on the hardware
- Most user programs require programs from OS – Input and Output
- Compiler builds calls to required system programs
- System programs must be found and linked
- Process of collecting system programs and linking them to user programs is called Linking and Loading or just Linking
- User + System = Load Module or Executable Image
- User programs must be linked to previously compiled user programs that reside in libraries -> Linker does this
- Speed of connection between a computer memory and processor determines the speed of computer
- Von Neumann Bottleneck -> Parallel Computers

Pure Interpretation





Pure Interpretation

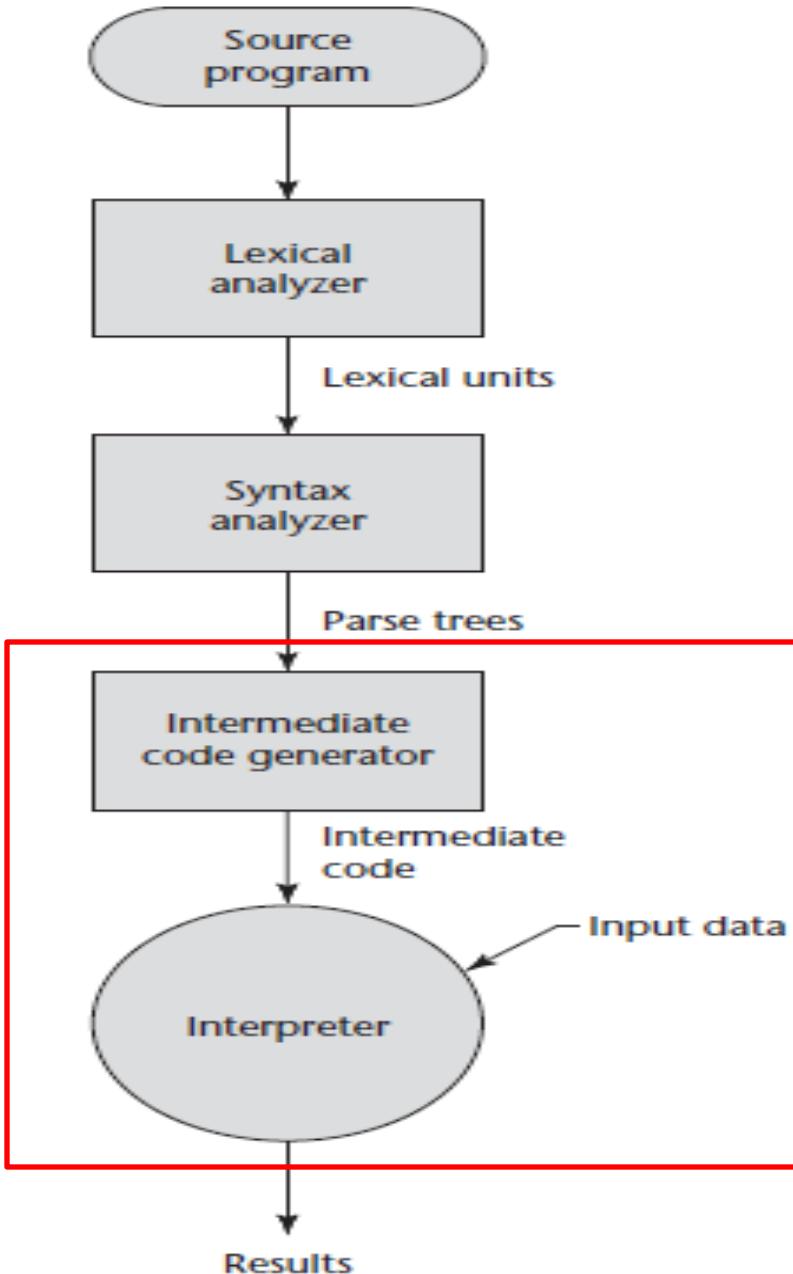
- Opposite to Compilation
- Programs are interpreted by another program called interpreter
- Interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high-level language programs
- Software simulation provides a virtual machine for the language
- **Adv:** Allows easy implementation of many source-level debugging operations
 - Run-time error messages can refer to source level units



Pure Interpretation

- **Disadv:** Execution is 10 to 100 times slower
 - Decoding of high-level language statements
- How many times a statement is executed, it must be decoded every time
- Statement decoding -> Bottleneck
- **Disadv:** Requires more storage space
 - Symbol table must be present during interpretation
 - Source program may be stored in a form designed for easy access and modification rather than one that provides minimal size
- **Eg:**
 - 1960 -> APL, SNOBOL, LISP
 - 1980 -> Rarely used on high-level languages
 - Recent years -> JS, PHP

Hybrid Implementation Systems





Hybrid Implementation Systems

- Compromise between compilers and pure interpreters
- Converts high-level language programs to an intermediate language designed to allow easy interpretation
- Faster -> Source language statements are decoded only once
- Instead of translating intermediate code to machine code, it interprets the code
- **Eg:** Perl -> Partially compiled to detect errors and to simplify interpreter
- Initial Java are hybrid -> intermediate form called byte code
 - Provides portability to any machine that has a byte code interpreter and an associated run-time system
 - Byte code interpreter + Run-time system = JVM



Hybrid Implementation Systems

- JIT
 - Translates program to intermediate language
 - Compiles intermediate language methods into machine code when they are called
 - Machine code version is kept for subsequent calls
 - JIT -> Java programs
 - .NET languages are all implemented with a JIT system
- Implementer provides both compiled and interpreted implementations for a language
 - Interpreter -> Develop and Debug programs
 - Bug-free state, programs are complied -> Execution speed ↑



Preprocessors

- Program that processes a program immediately before compilation
- Preprocessor instructions are embedded in programs
 - Used to specify that code from another file is to be included
- Preprocessor -> Macro expander

#include myLib.c

- Causes the preprocessor to copy the contents of “myLib.c” into the program at the position “#include”



Preprocessors

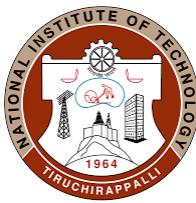
```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

- **Expression:** $x = \max(2*y, z/1.73);$

```
x = ((2*y) > (z/1.73)? (2*y) : (z/1.73);
```



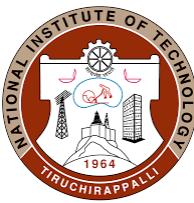
Thank You



CSPC31: Principles of Programming Languages

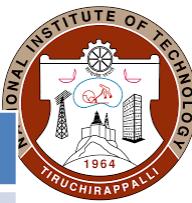
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 3 – Describing Syntax and Semantics



Objectives

- Define – Syntax and Semantics
- Common method of describing syntax -> Context-free grammars (also known as Backus-Naur Form)
- Derivations, Parse Trees, Ambiguity, Descriptions of Operator Precedence and Associativity, Extended Backus-Naur Form
- Attribute grammars, which can be used to describe both the syntax and static semantics of programming languages
- Three formal methods of describing semantics
 - Operational, Axiomatic, Denotational Semantics



Introduction

- Providing a concise and understandable description of a programming language is difficult but essential
- ALGOL 60 and ALGOL 68 -> Descriptions were not easily understandable
- **Pbm in describing a language:** Diversity of the audience
 - Initial Evaluators, Implementors, Production Users
- Initial Evaluators -> Users within the organization
- Implementers -> Determine how the expressions, statements and program units are formed and also their effect when executed
 - Difficulty of the job of implementers -> Completeness and precision of the language description



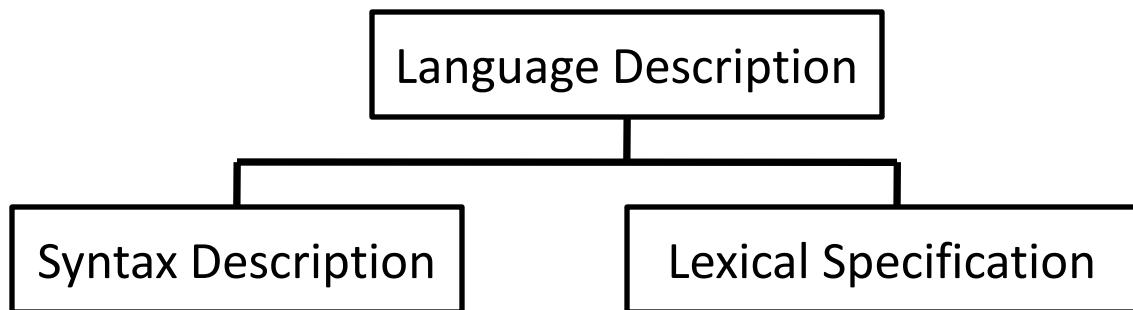
Introduction

- Production Users -> Determine how to encode software solutions by referring to a language reference model
 - Textbooks and Courses -> But language manuals are usually the only authoritative printed information source about a language
- Study of programming language -> Syntax and Semantics
 - Syntax -> Form of expressions, statements and program units
 - Semantics -> Meaning of expressions, statements and program units
- Eg: `while (<boolean_expr>) <statement>`
 - Semantics -> When the current value of Boolean expression is true, the statement is executed
- Semantics should follow directly from syntax
- Appearance of a statement should strongly suggest what the statement is meant to accomplish



General Problem of Describing Syntax

- Language -> Set of strings of characters from some alphabets
- Strings of a language are called sentences or statements
- Syntax rules of a language specify which strings of characters from language's alphabet are in the language
- Formal descriptions of the syntax do not include descriptions of the lowest-level syntactic units -> Lexemes
- Description of lexemes can be given by a lexical specification, which is usually separate from the syntactic description





General Problem of Describing Syntax

- Lexemes includes numeric literals, operators, special words, etc.
 - A numeric literal is a character-string whose characters are selected from the digits 0 through 9, a sign character (+ or -), and the decimal point. If the literal contains no decimal point, it is an integer
- One can think of programs as strings of lexemes rather than of characters
- Lexemes are partitioned into groups
 - Names of variables, methods, classes, etc. -> Identifiers
- Each lexeme group is represented by a name or token
- Token of a language is a category of lexemes
 - **Eg. 1:** Identifier is a token that can have lexemes or instances such as sum, total, etc.
 - **Eg. 2:** Token for arithmetic operator symbol "+" has only one possible lexeme (plus_op)



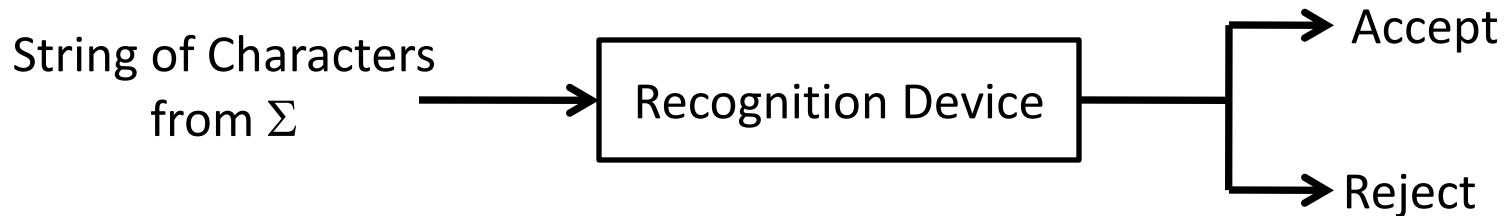
General Problem of Describing Syntax

Index = 2 * count + 17;

Lexemes	Tokens
Index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Language Recognizers

- Languages can be defined in 2 ways -> 1. By Recognition;
2. By Generation
- Eg: Language L that uses an alphabet Σ of characters



- If R, when fed any strings of characters over Σ , accepts it only if it is in L, then **R is a description of L**
- Recognition devices are not used to enumerate all of the sentences of a language
- Syntax analysis part of compiler is the recognizer for the language the compiler translates
- Syntax analyzer determines whether the given programs are in the language and are syntactically correct as well



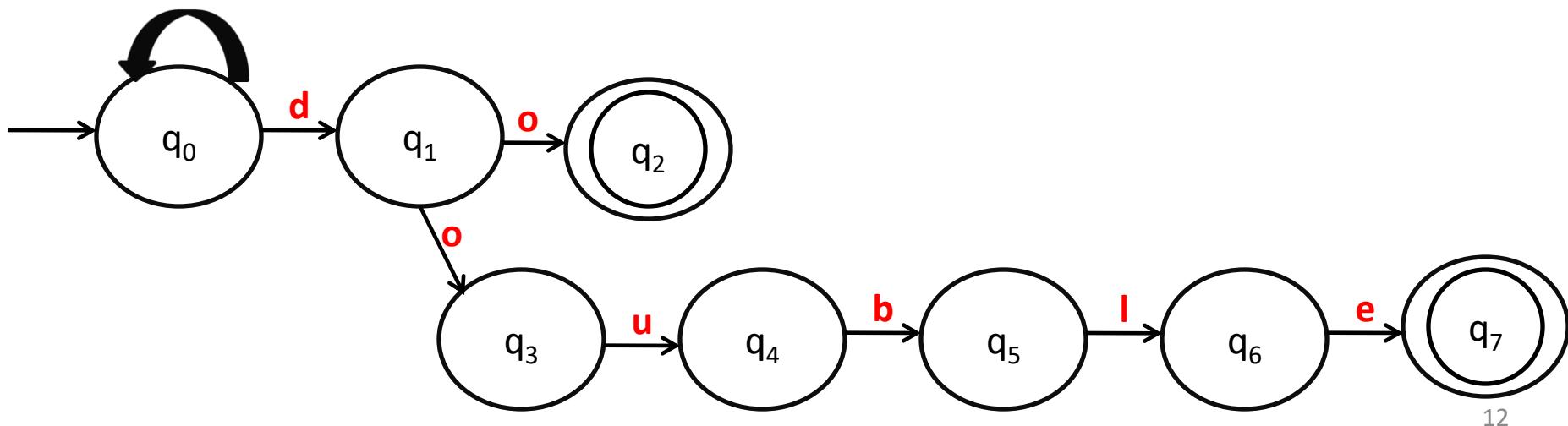
Non-Deterministic Finite Automata (NFA)- Language Recognizer

$$\Sigma = \{a, b, c, \dots, z\}$$

Input: **double**

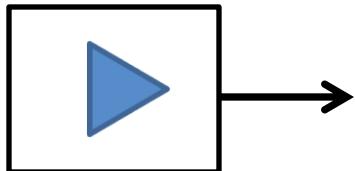
Valid keywords of my programming language:
do, double

a, b, c,, z (Except d)



Language Generators

- Device that can be used to generate the sentences of a language



When pushed generates
a Sentence

- Sentence produced by a generator is unpredictable
- People prefer certain forms of generators over recognizers
- To determine the correct syntax of a particular statement using a compiler:
 - Recognizers -> Trial-and-error mode
 - Generators -> Compare with the structure of the generator
- Close connection between formal generation and recognition devices for the same language
 - One of the seminal discoveries in computer science, and it led to much of what is now known about formal languages and compiler design theory



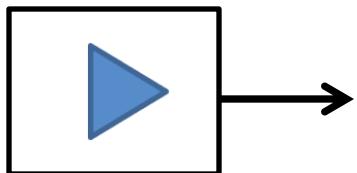
Regex - Language Generator

Regular Expression = $a(a^* \cup b^*)b$

* -> Means 0 or more occurrence

- Description:
 - First output an “a”. Then do one of the following two things:
 - Either output a number of a’s or output a number of b’s
 - Finally output a “b”
- Various Strings generated by the given Regular Expression:

ab , aab , $aaab$, $aaaab$, $aaa\ldots b$, abb , $abbb$, $abbbb$, $abb\ldots b$



When pushed generates a string with first letter “a” and last letter “b”

Generated String: ab
Input in Hand: $aaab$



Regex - Language Generator

Regular Expression = $a(a^* \cup b^*)b$

* -> Means 0 or more occurrence

- Description:
 - First output an “a”. Then do one of the following two things:
 - Either output a number of a’s or output a number of b’s
 - Finally output a “b”
- Various Strings generated by the given Regular Expression:
 ab , aab , $aaab$, $aaaab$, $aaa....b$, abb , $abbb$, $abbbab$, $abb....b$
- Grammar for the given Regular Expression is:

$$S \rightarrow aMb$$

$$M \rightarrow A \mid B$$

$$A \rightarrow \epsilon \mid aA$$

$$B \rightarrow \epsilon \mid bB$$

Input in Hand: $aaab$



Formal Methods of Describing Syntax

- Formal language generation mechanisms called grammars are used to describe the syntax
- Backus-Naur Form and Context-Free Grammars
 - Middle to late 1950s, Noam Chomsky and John Backus developed the same syntax description formalism
 - Most widely used method for programming language syntax



Context-free Grammars

- Chomsky described 4 classes of generative devices or grammars that define 4 classes of languages
- Two of these grammar classes, named Context-free and Regular -> Useful for describing the syntax of programming languages
 - Regular grammars -> Describes the forms of tokens
 - Context-free grammars -> Syntax (with minor exceptions)



Origins of Backus-Naur Form

- ACM-GAMM group began designing ALGOL 58
- John Backus presented paper in Int'l Conf in 1959
- Introduced new formal notation for specifying programming language syntax
- Later modified by Peter Naur for ALGOL 60
- Revised method of syntax description -> Backus-Naur Form or BNF
- BNF is a natural notation for describing syntax of many natural languages
- Most popular method of concisely describing programming language syntax
- Remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammars



Miscellaneous

total = subtotal1 + subtotal2

<assign> → stmt

<stmt> → <var> = <expression>

<var> → total | subtotal1 | subtotal2

<expression> → <var> + <var>



Fundamentals

- Metalanguage -> Used to describe another language
- BNF uses abstraction for syntactic structures
- Java assignment statement represented by <assign>
 - Definition: <assign> → <var> = <expression>
- Symbol on LHS of the arrow is the abstraction being defined
- Text on RHS is the definition of LHS – mixtures of tokens, lexemes and references to other abstractions
- Altogether, the definition is called a rule or production
- Abstractions <var> and <expression> must be defined for the <assign> definition to be useful

total = subtotal1 + subtotal2



Fundamentals

- Abstractions in BNF or grammar -> non-terminals
- Lexemes and Tokens of the rules -> terminals
- BNF description or grammar is simply a collection of rules
- Non-terminal symbols can have two or more distinct definitions -> representing two or more possible syntactic forms in the language

$\langle \text{if_stmt} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$

$\langle \text{if_stmt} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

(or)

$\langle \text{if_stmt} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \mid$

$\text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- Describe lists of similar constructs, order in which different constructs must appear, nested structures to any depth, imply operator precedence and operator associativity



Describing Lists

- Variable-length lists are written using (1, 2, ...)
- BNF requires an alternate method

int a, b, c;

- Uses Recursion
- A rule is recursive if its LHS appears in its RHS

$\text{<ident_list>} \rightarrow \text{identifier} \mid$
 $\text{identifier, <ident_list>}$

- $\text{<ident_list>} \rightarrow$ either a single token (identifier) or an identifier followed by a comma followed by another instance of <ident_list>



Grammars and Derivations

- Grammar is a generative device for defining languages
- Sentences are generated through a sequence of applications of rules, beginning with the start symbol
- Sentence generation is called a derivation



Grammars and Derivations

Grammar for a small language

```
<program> → begin <stmt_list> end  
<stmt_list> → <stmt>  
          | <stmt> ; <stmt_list>  
<stmt> → <var> = <expression>  
<var> → A | B | C  
<expression> → <var> + <var>  
           | <var> - <var>  
           | <var>
```

```
<program> => begin <stmt_list> end  
=> begin <stmt> ; <stmt_list> end  
=> begin <var> = <expression> ; <stmt_list> end  
=> begin A = <expression> ; <stmt_list> end  
=> begin A = <var> + <var> ; <stmt_list> end  
=> begin A = B + <var> ; <stmt_list> end  
=> begin A = B + C ; <stmt_list> end  
=> begin A = B + C ; <stmt> end  
=> begin A = B + C ; <var> = <expression> end  
=> begin A = B + C ; B = <expression> end  
=> begin A = B + C ; B = <var> end  
=> begin A = B + C ; B = C end
```

Generated Sentence
24



Grammars and Derivations

- <program> is the Start symbol
- => means “derives”
- Each successive string is derived from the previous string
- Each string in the derivation is called a sentential form
- Leftmost Derivation -> Leftmost non-terminal will be replaced
 - Rightmost, Both Left and Right also exists
- Derivation continues until no non-terminals are left
- Sentential form containing only terminals or lexemes -> Generated Sentence



Grammars and Derivations

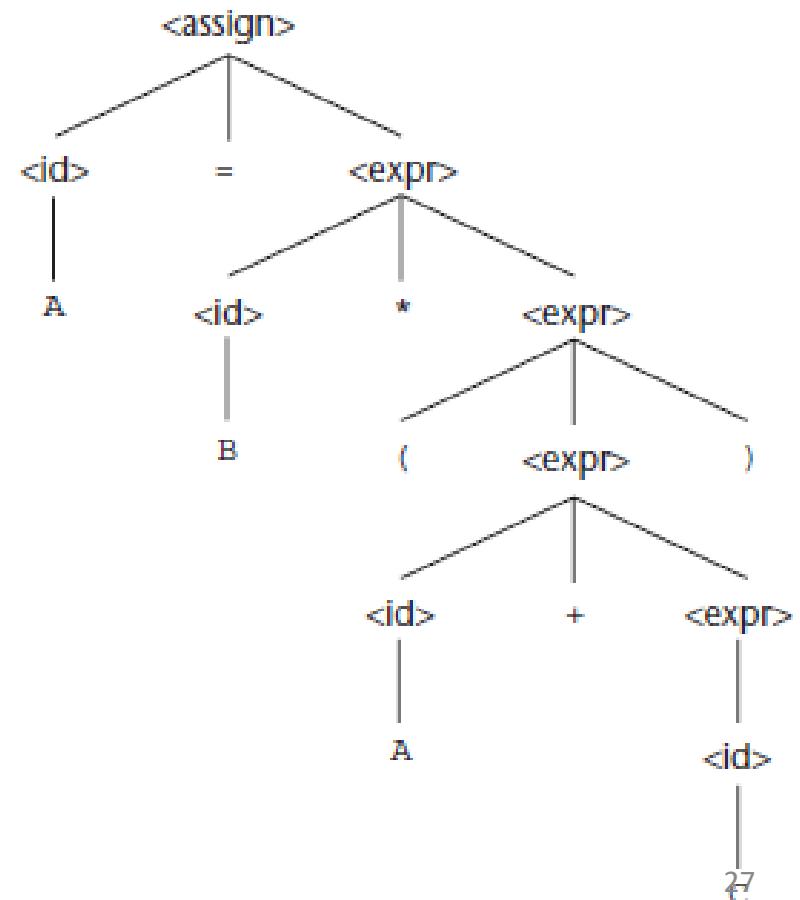
Grammar for Simple Assignment Statement: $A = B * (A + C)$

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
         | <id> * <expr>
         | ( <expr> )
         | <id>
```

```
<assign> => <id> = <expr>
          => A = <expr>
          => A = <id> * <expr>
          => A = B * <expr>
          => A = B * ( <expr> )
          => A = B * ( <id> + <expr> )
          => A = B * ( A + <expr> )
          => A = B * ( A + <id> )
          => A = B * ( A + C )
```

Parse Trees

- One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the language
- Hierarchical structures -> Parse Trees
- Every internal node is labelled with a non-terminal symbol
- Every leaf is labelled with a terminal symbol
- Every sub-tree describes one instance of an abstraction in the sentence

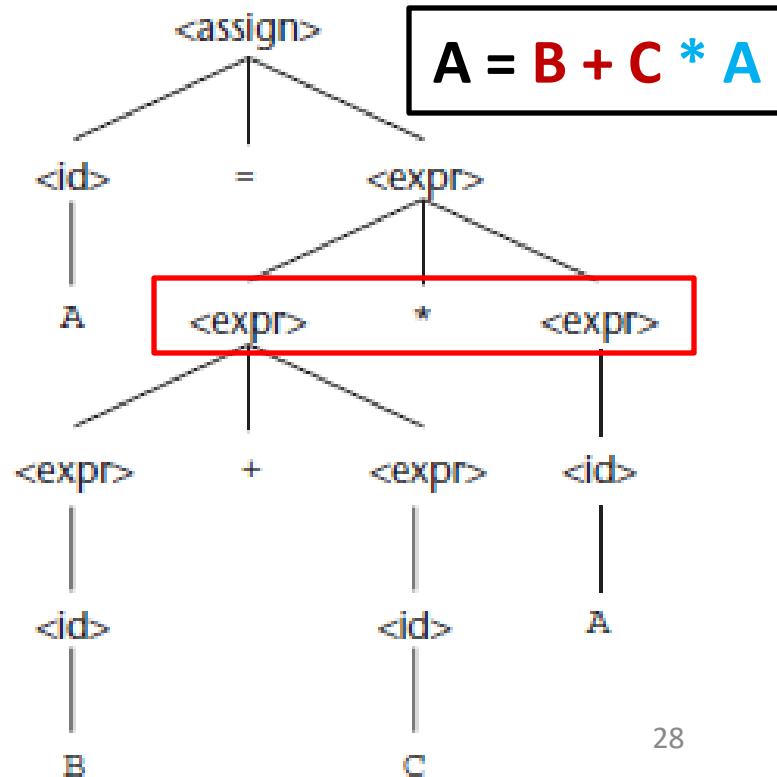
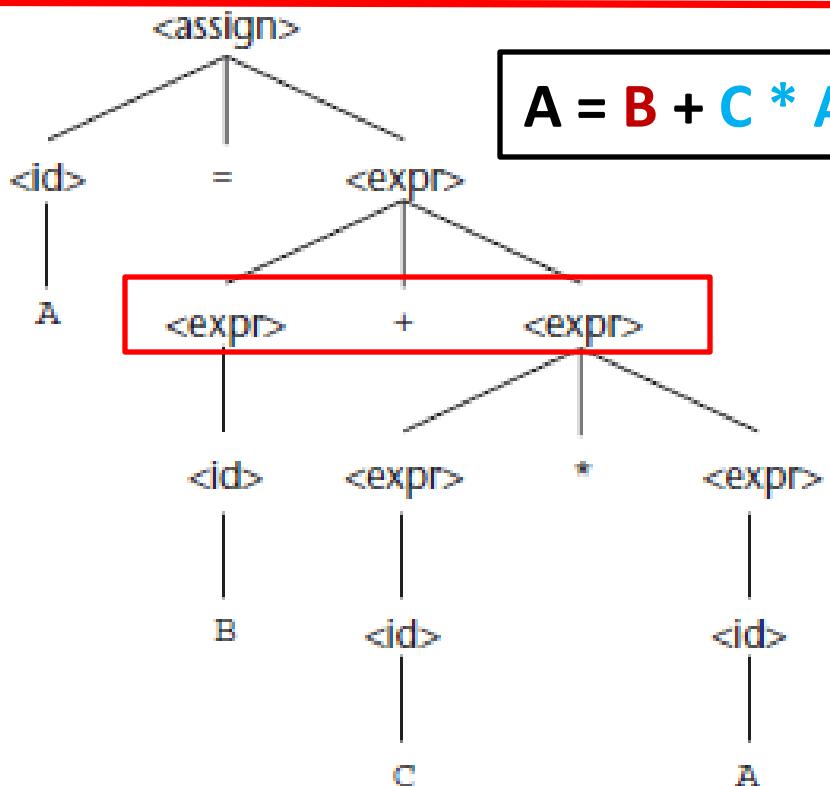


Ambiguity

- Grammar that generates a sentential form for which two or more distinct parse tree exists

$$\begin{aligned}
 <\text{assign}> &\rightarrow <\text{id}> = <\text{expr}> \\
 <\text{id}> &\rightarrow A \mid B \mid C \\
 <\text{expr}> &\rightarrow <\text{expr}> + <\text{expr}> \\
 &\quad | <\text{expr}> * <\text{expr}> \\
 &\quad | (<\text{expr}>) \\
 &\quad | <\text{id}>
 \end{aligned}$$

A = B + C * A





Ambiguity

- Ambiguity occurred -> Grammar specifies slightly less syntactic structure
- Allows the parse tree to grow both on the right and the left
- Syntactic ambiguity is a problem -> Compilers often base the semantics of those structures on their syntactic form
 - Code generation relies on the information in parse tree
- Characteristics to determine ambiguity
 - Grammar generates a sentence with more than one leftmost derivation
 - Grammar generates a sentence with more than one rightmost derivation
 - More than one parse tree
- Some parsing algorithms can be based on ambiguous grammars
 - Uses non-grammatical information provided by the designer



Operator Precedence

- Expression having two different operators $\rightarrow X + Y * Z$
 - Order of evaluation of the operators
- Assign different precedence level
- Grammar can describe certain syntactic structure – Meaning of the structure can be determined from its parse tree
 - More lower an operator in an arithmetic expression appears in a parse tree, the higher priority it has
- **Soln:** Add additional non-terminals and some new rules
 - $\langle \text{term} \rangle, \langle \text{factor} \rangle$



Operator Precedence

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \underline{\langle \text{expr} \rangle + \langle \text{term} \rangle}$
| $\langle \text{term} \rangle$

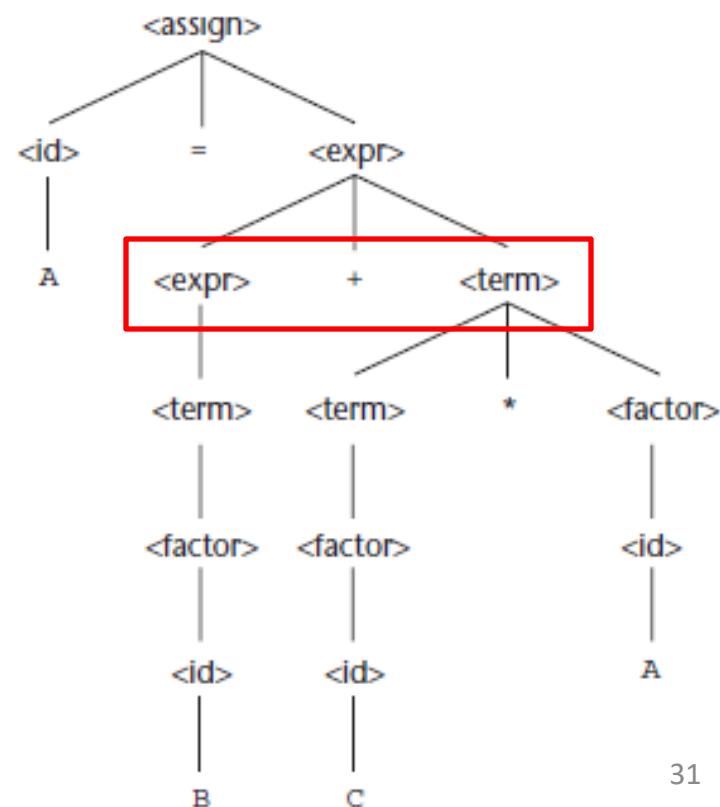
$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
| $\langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$
| $\langle \text{id} \rangle$

A = B + C * A

Leftmost Derivation

$\langle \text{assign} \rangle \Rightarrow \underline{\langle \text{id} \rangle} = \langle \text{expr} \rangle$
 $\Rightarrow A = \underline{\langle \text{expr} \rangle}$
 $\Rightarrow A = \underline{\langle \text{expr} \rangle + \langle \text{term} \rangle}$
 $\Rightarrow A = \underline{\langle \text{term} \rangle + \langle \text{term} \rangle}$
 $\Rightarrow A = \underline{\langle \text{factor} \rangle + \langle \text{term} \rangle}$
 $\Rightarrow A = \underline{\langle \text{id} \rangle + \langle \text{term} \rangle}$
 $\Rightarrow A = B + \langle \text{term} \rangle$
 $\Rightarrow A = B + \underline{\langle \text{term} \rangle * \langle \text{factor} \rangle}$
 $\Rightarrow A = B + \underline{\langle \text{factor} \rangle * \langle \text{factor} \rangle}$
 $\Rightarrow A = B + \underline{\langle \text{id} \rangle * \langle \text{factor} \rangle}$
 $\Rightarrow A = B + C * \underline{\langle \text{factor} \rangle}$
 $\Rightarrow A = B + C * \underline{\langle \text{id} \rangle}$
 $\Rightarrow A = B + C * A$





Operator Precedence

- Close connection between parse trees and derivations
- Either can easily be constructed from the other
- Every derivation with an unambiguous grammar has a unique parse tree although that tree can be represented by different derivations
 - Leftmost Derivation & Rightmost Derivation

Operator Precedence

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \underline{\langle \text{expr} \rangle + \langle \text{term} \rangle}$
 $\quad \mid \langle \text{term} \rangle$

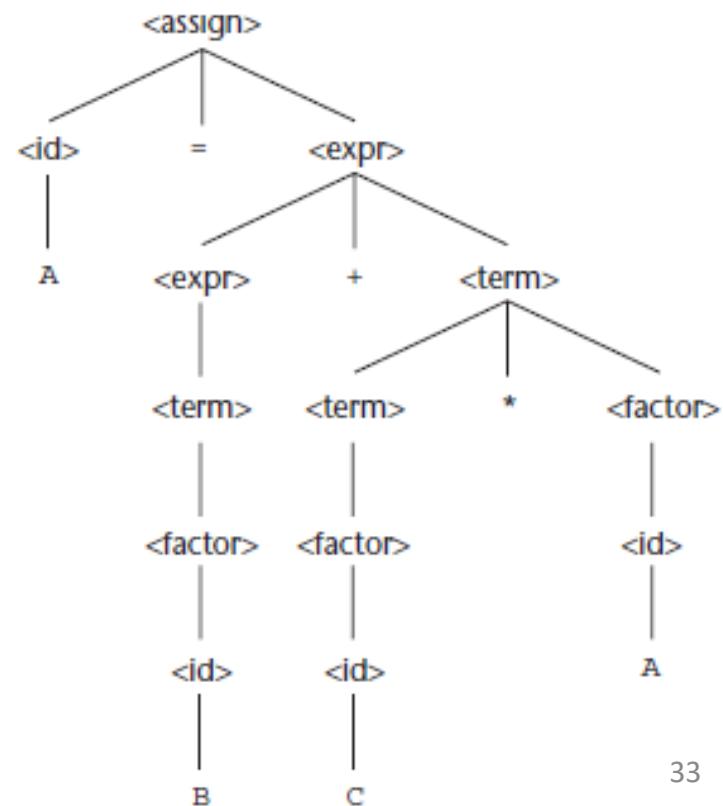
$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\quad \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$
 $\quad \mid \langle \text{id} \rangle$

A = B + C * A

Rightmost Derivation

$\langle \text{assign} \rangle \Rightarrow \langle \text{id} \rangle = \underline{\langle \text{expr} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{term} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \langle \text{term} \rangle * \underline{\langle \text{factor} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \langle \text{term} \rangle * \underline{\langle \text{id} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{term} \rangle * A}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{factor} \rangle * A}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{id} \rangle * A}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + C * A$
 $\Rightarrow \langle \text{id} \rangle = \underline{\langle \text{term} \rangle} + C * A$
 $\Rightarrow \langle \text{id} \rangle = \underline{\langle \text{factor} \rangle} + C * A$
 $\Rightarrow \langle \text{id} \rangle = \underline{\langle \text{id} \rangle} + C * A$
 $\Rightarrow \underline{\langle \text{id} \rangle} = B + C * A$
 $\Rightarrow \underline{A} = B + C * A$





Associativity Of Operators

- Expression involving two operators with same precedence
 - Semantic rule is required to specify the preference → Associativity
- In most cases, associativity of addition in a computer is irrelevant

$$(A + B) + C = A + (B + C)$$

- Floating-point addition in a computer is not necessarily associative
- Correct associativity may be essential for expressions that have such issues



Floating Point Addition

$$1.0 \times 10^7 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$$

Left Associative

$$1 \times 10^7 = 1.0000000 \times 10^7$$

$$\begin{array}{r} 1 = 1 \times 10^0 = 0.0000001 \times 10^7 \\ + \\ \hline 1.0000001 \times 10^7 \end{array}$$

$$1 \times 10^7 = 1.0000000 \times 10^7$$

$$\begin{array}{r} 1 = 1 \times 10^0 = 0.0000001 \times 10^7 \\ + \\ \hline 1.0000001 \times 10^7 \end{array}$$

.

Store seven Digits of Accuracy = 1.000000×10^7

Right Associative

$$1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 10$$

$$10 = 1 \times 10^1 = 0.0000010 \times 10^7$$

$$\begin{array}{r} 1 \times 10^7 = 1.0000000 \times 10^7 \\ + \\ \hline 1.0000010 \times 10^7 \end{array}$$

Store seven Digits of Accuracy = 1.000001×10^7



Associativity of Operators

$$A = B + C + A$$

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\quad \mid \langle \text{term} \rangle$

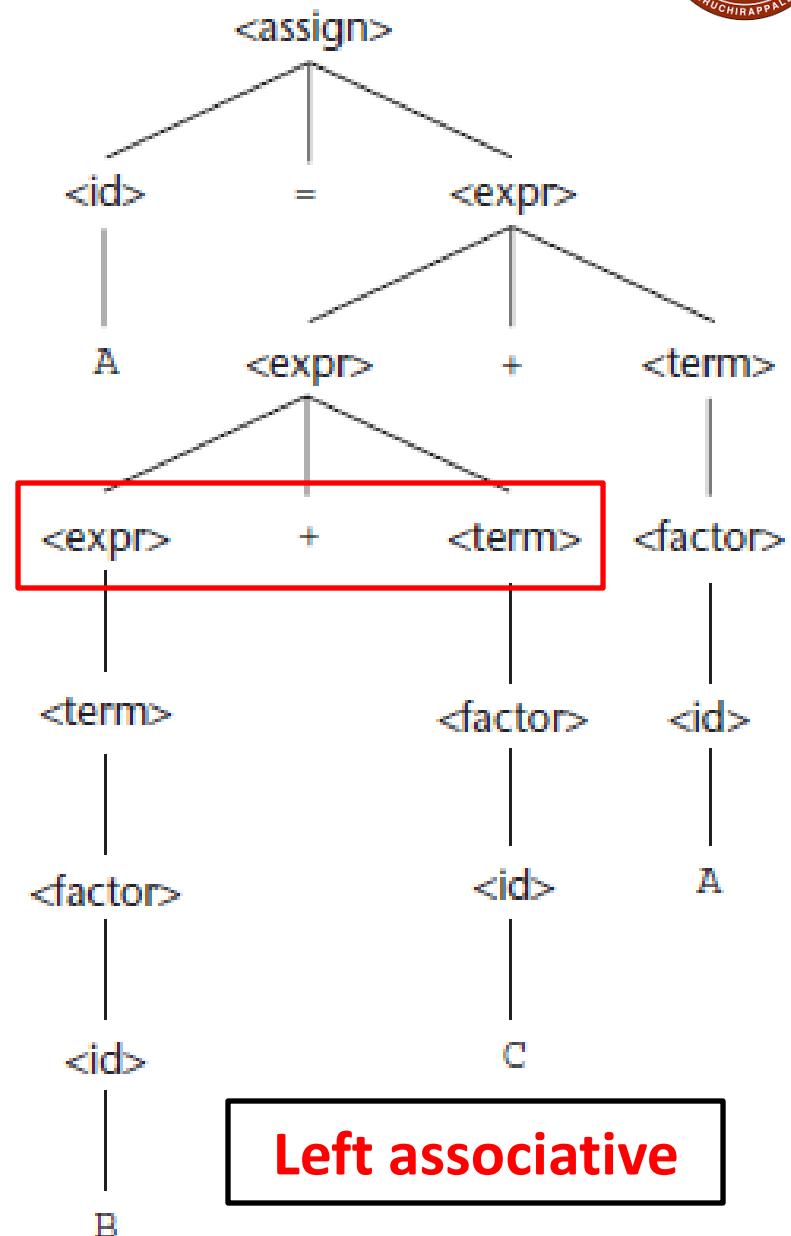
$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\quad \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$
 $\quad \mid \langle \text{id} \rangle$

- Left Recursive -> Grammar rule has its LHS also appearing at the beginning of its RHS

- Specifies Left Associativity ->
Statement will be executed from
Left to Right

- Disallows the use of some
important syntax analysis
algorithms





Associativity of Operators

- Exponentiation operator is right associative
 - Use right recursion

$$\begin{aligned}\text{<factor>} &\rightarrow \text{<exp>} \boxed{\text{**}} \text{ <factor>} \\ &\quad | \text{ <exp>} \\ \text{<exp>} &\rightarrow (\text{ <expr>}) \\ &\quad | \text{id}\end{aligned}$$

- Used to describe exponentiation as a right-associative operator -> Statement will be executed from Right to Left

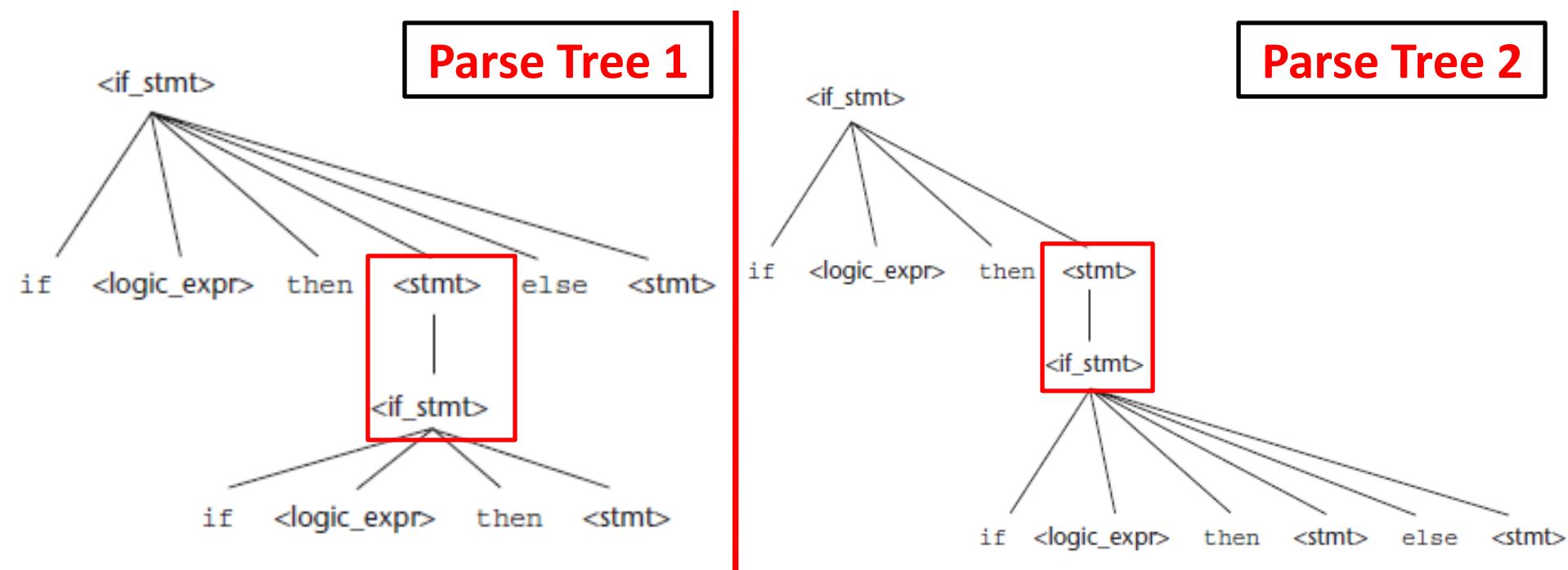
Unambiguous Grammar for if-then-else

$\langle \text{if_stmt} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$

$\text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- Adding $\langle \text{stmt} \rangle \rightarrow \langle \text{if_stmt} \rangle$ creates ambiguity

If $\langle \text{logic_expr} \rangle \text{ then if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$



Unambiguous Grammar for if-then-else



$\langle \text{if_stmt} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$
| $\text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$
 $\langle \text{stmt} \rangle \rightarrow \langle \text{if_stmt} \rangle$

If $\langle \text{logic_expr} \rangle \text{ then if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

Derivation 1:

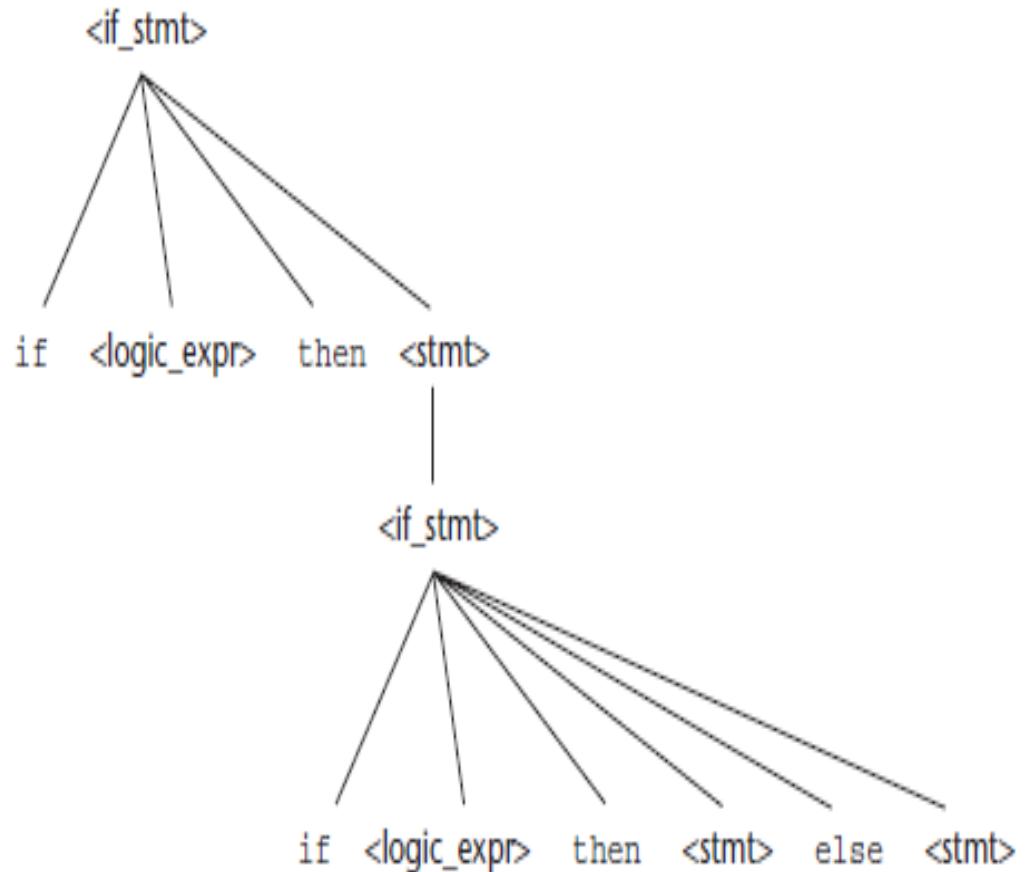
$\langle \text{if_stmt} \rangle \Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$
 $\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{if_stmt} \rangle$
 $\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \rangle$

Derivation 2:

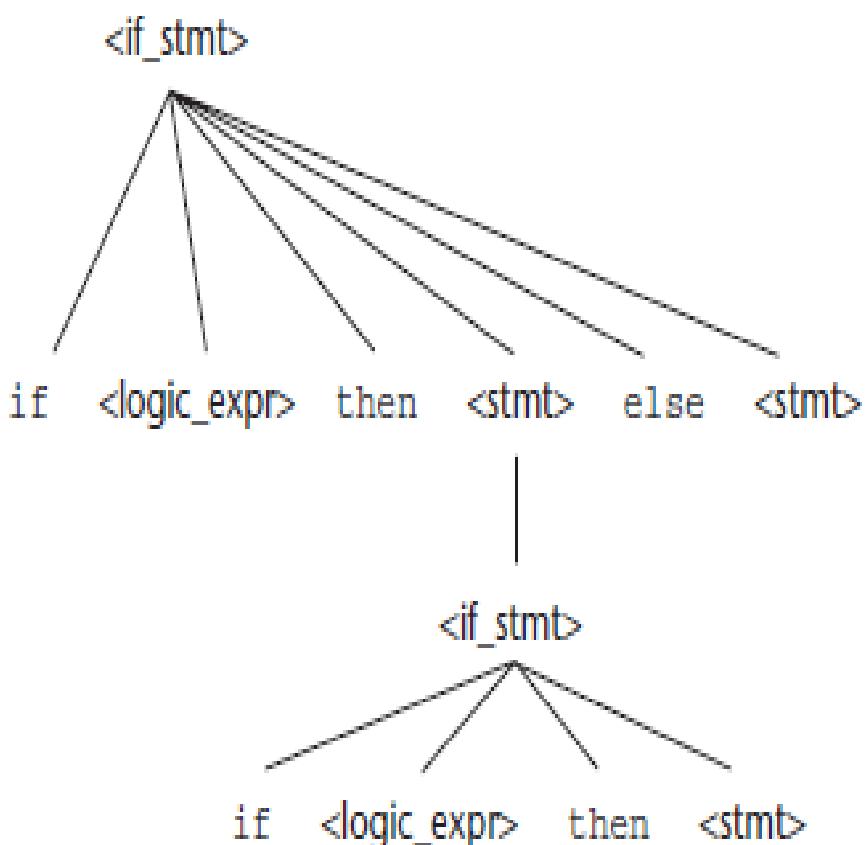
$\langle \text{if_stmt} \rangle \Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$
 $\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{if_stmt} \rangle \text{ else } \langle \text{stmt} \rangle$
 $\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \rangle$

Unambiguous Grammar for if-then-else

Parse Tree 1



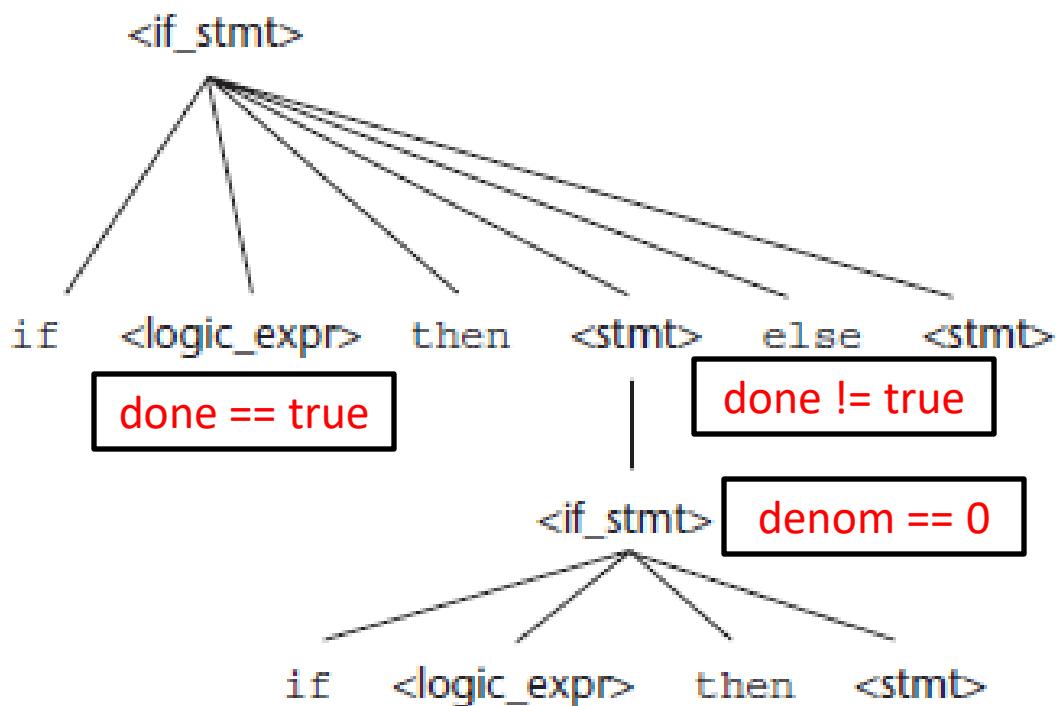
Parse Tree 2



Unambiguous Grammar for if-then-else

```

if done == true
  then if denom == 0
    then quotient = 0;
  else quotient = num / denom;
  
```



Unambiguous Grammar for if-then-else

- **Pbm:** Treats all statements as if they are all matched
- Rule for **if** constructs is that an **else**, when present, is matched with the nearest previous unmatched **then**
- Unmatched statements -> **else-less ifs**
- Matched Statements -> All other statements
- Different abstractions or non-terminals must be used



Unambiguous Grammar for if-then-else

$\langle \text{stmt} \rangle \rightarrow \langle \text{matched} \rangle \mid \langle \text{unmatched} \rangle$

$\langle \text{matched} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle$
| any non-if statement

$\langle \text{unmatched} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$
| $\text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{unmatched} \rangle$

$\text{if } \langle \text{logic_expr} \rangle \text{ then } \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

Derivation 1:

$\langle \text{stmt} \rangle \Rightarrow \langle \text{unmatched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then if } \langle \text{logic_expr} \rangle \text{ then } \text{any non-if statement} \text{ else } \langle \text{matched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{statement} \rangle \text{ else } \langle \text{matched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{statement} \rangle \text{ else } \text{any non-if statement}$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{statement} \rangle \text{ else } \langle \text{statement} \rangle$

Derivation 2:

$\langle \text{stmt} \rangle \Rightarrow \langle \text{matched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \text{any non-if statement} \text{ else } \langle \text{matched} \rangle$

$\Rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle$

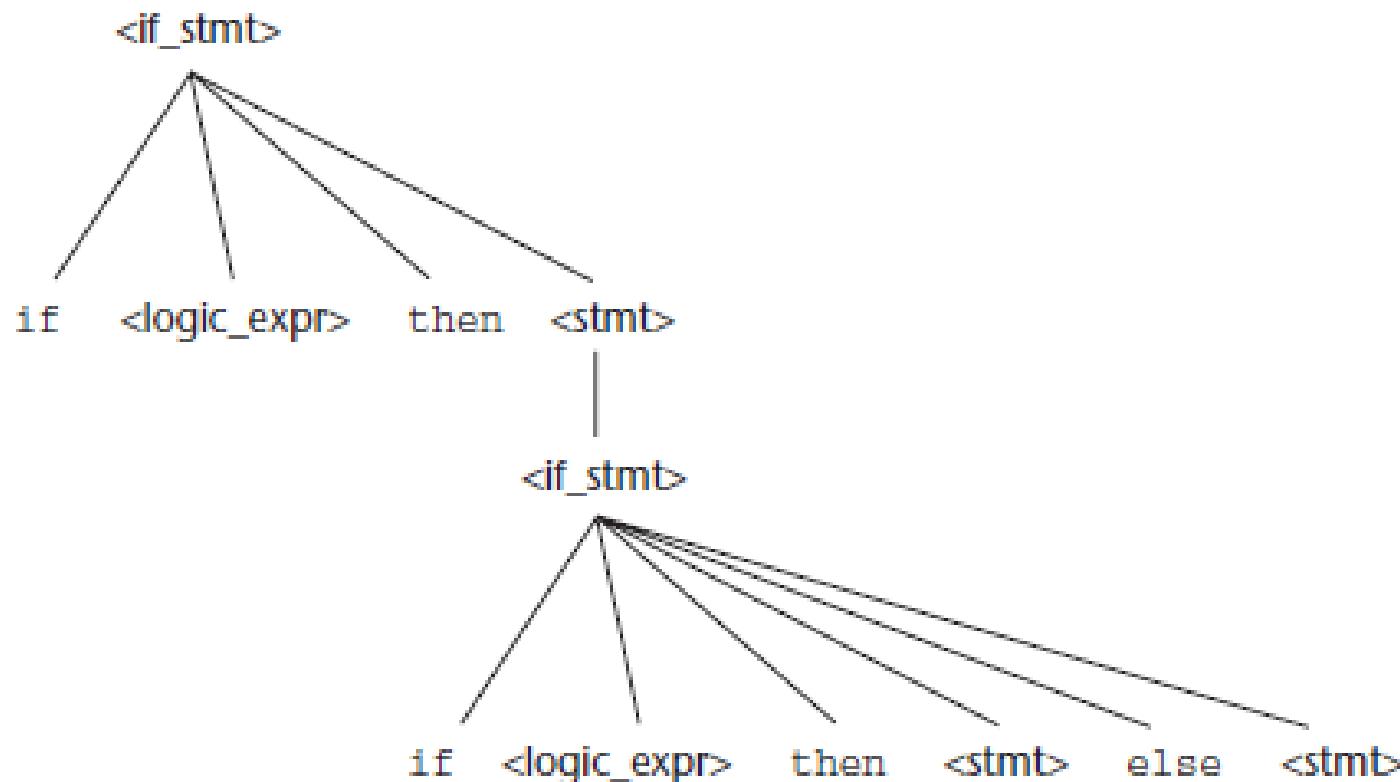
Unambiguous Grammar for if-then-else

$\langle \text{stmt} \rangle \rightarrow \langle \text{matched} \rangle \mid \langle \text{unmatched} \rangle$

$\langle \text{matched} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle$
 | any non-if statement

$\langle \text{unmatched} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$
 | $\text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{unmatched} \rangle$

if <logic_expr> then if <logic_expr> then <stmt> else <stmt>





Extended BNF

- Extension of BNF -> Extended BNF or EBNF
 - Do not enhance the descriptive power of BNF
 - Increase readability and writability
- Three extensions
 - Optional part of an RHS
 - Indefinite repetition or Leave it altogether
 - Multiple-choice options



Extended BNF

Extension 1: Optional part of the RHS

<selection> → if (<expression>) <statement>
| if (<expression>) <statement> else <statement>
<selection> → if (<expression>) <statement> [else <statement>]

Extension 2: Indefinite repetition or Leave it altogether

<ident_list> → <identifier> {, <identifier>}

Extension 3: Multiple choice options

<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor>
<term> → <term> (* | / | %) <factor>



Extended BNF

- Brackets, braces and parantheses -> Metasymbols
 - Notational tools and not terminal symbols
- $\textcolor{red}{<\text{expr}> \rightarrow <\text{expr}> + <\text{term}>}$ => **Left Associative**
- $\textcolor{purple}{<\text{expr}> \rightarrow <\text{term}> \{+ <\text{term}> \}}$ => **Does not imply Left Associativity**
- Various versions exist
- Numeric superscript or “+” symbol attached to right brace-> Indicate an upper limit
 - $\textcolor{red}{<\text{compound}> \rightarrow \text{begin } <\text{stmt}> \{<\text{stmt}> \} \text{ end}}$
 - $\textcolor{purple}{<\text{compound}> \rightarrow \text{begin } \{<\text{stmt}> \}^+ \text{ end}}$

“+” means 1 or more occurrence



Extended BNF

- Replace arrow by colon and place RHS on next line

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$

$\langle \text{expr} \rangle :$

$\langle \text{expr} \rangle + \langle \text{term} \rangle$

- No need to place vertical bars to separate alternative RHSs

$\langle \text{if_stmt} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$

 | $\text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

$\langle \text{if_stmt} \rangle \rightarrow \text{if } \langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle$

 if $\langle \text{logic_expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$



Extended BNF

- Indicate optional using subscript “opt”

`<selection> → if (<expression>) <statement> [else <statement>]`

`<selection> → if (<expression>) <statement> (else <statement>opt)`

- Instead of “|” use “one of” keyword

`AssignmentOperator → = | *= | /= | %=`

`AssignmentOperator → one of = *= /= %=`

- Standard for EBNF (ISO/IEC 14977:1996(1996))

- Use equal sign instead of an arrow in rules
- Terminate each RHS with a semi-colon
- Use quotes on all terminal symbols



Grammars and Recognizers

- Close relationship between generation and recognition devices for a given language
- Given a context-free grammar, a recognizer for the language generated by the grammar can be automatically constructed
- Software systems already exist
 - Allows quick creation of the syntax analysis part of a compiler for a new language
 - Well-known compiler (yacc -> yet another compiler-compiler)

Miscellaneous

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \underline{\langle \text{expr} \rangle + \langle \text{term} \rangle}$
 $\quad \mid \langle \text{term} \rangle$

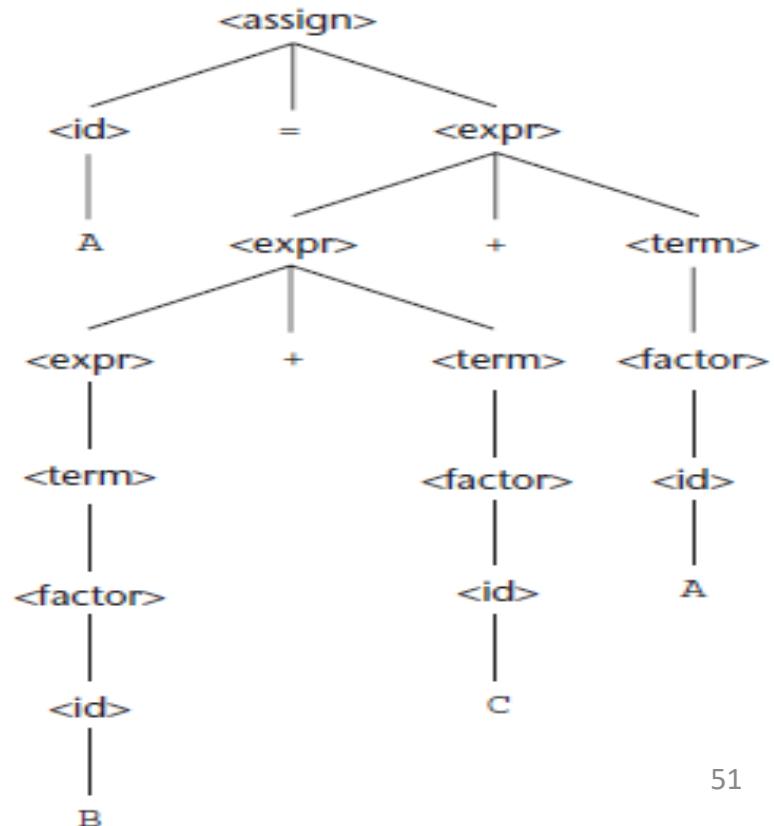
$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\quad \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$
 $\quad \mid \langle \text{id} \rangle$

A = B + C + A

Rightmost Derivation

$\langle \text{assign} \rangle \Rightarrow \langle \text{id} \rangle = \underline{\langle \text{expr} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{term} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \langle \text{term} \rangle + \underline{\langle \text{factor} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \langle \text{term} \rangle + \underline{\langle \text{id} \rangle}$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{term} \rangle} + A$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{factor} \rangle} + A$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + \underline{\langle \text{id} \rangle} + A$
 $\Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle + C + A$
 $\Rightarrow \langle \text{id} \rangle = \underline{\langle \text{term} \rangle} + C + A$
 $\Rightarrow \langle \text{id} \rangle = \underline{\langle \text{factor} \rangle} + C + A$
 $\Rightarrow \langle \text{id} \rangle = \underline{\langle \text{id} \rangle} + C + A$
 $\Rightarrow \langle \text{id} \rangle = B + C + A$
 $\Rightarrow \underline{A} = B + C + A$





Miscellaneous

```
Expr → Expr + Term  
Expr → Term  
Term → Term * Factor  
Term → Factor  
Factor → "(" Expr ")"  
Factor → integer
```

Objective: Track the value computed at each stage

Non-Terminals:

- Expr
- Term
- Factor

Attribute:
value

```
Expr1 → Expr2 + Term [ Expr1.value = Expr2.value + Term.value ]  
Expr → Term [ Expr.value = Term.value ]  
Term1 → Term2 * Factor [ Term1.value = Term2.value * Factor.value ]  
Term → Factor [ Term.value = Factor.value ]  
Factor → "(" Expr ")" [ Factor.value = Expr.value ]  
Factor → integer [ Factor.value = strToInt(integer.str) ]
```



Miscellaneous

$\langle \text{proc_def} \rangle \rightarrow \text{procedure } \langle \text{proc_name} \rangle [1]$
 $\quad \quad \quad \langle \text{proc_body} \rangle \text{ end } \langle \text{proc_name} \rangle [2] ;$

Non-Terminal:

- proc_name

Attribute:

string

Objective: Identify the beginning and ending of procedure / function

Syntax rule: $\langle \text{proc_def} \rangle \rightarrow \text{procedure } \underline{\langle \text{proc_name} \rangle [1]}$
 $\quad \quad \quad \underline{\langle \text{proc_body} \rangle \text{ end }} \underline{\langle \text{proc_name} \rangle [2]} ;$

Predicate: $\langle \text{proc_name} \rangle [1] \underline{\text{string}} == \langle \text{proc_name} \rangle [2] . \underline{\text{string}}$

Notice that when there is more than one occurrence of a nonterminal in a syntax rule in an attribute grammar, the nonterminals are subscripted with brackets to distinguish them. Neither the subscripts nor the brackets are part of the described language.

Miscellaneous

A = A + B

<assign> → <var> = <expr>

<expr> → <var> + <var>

| <var>

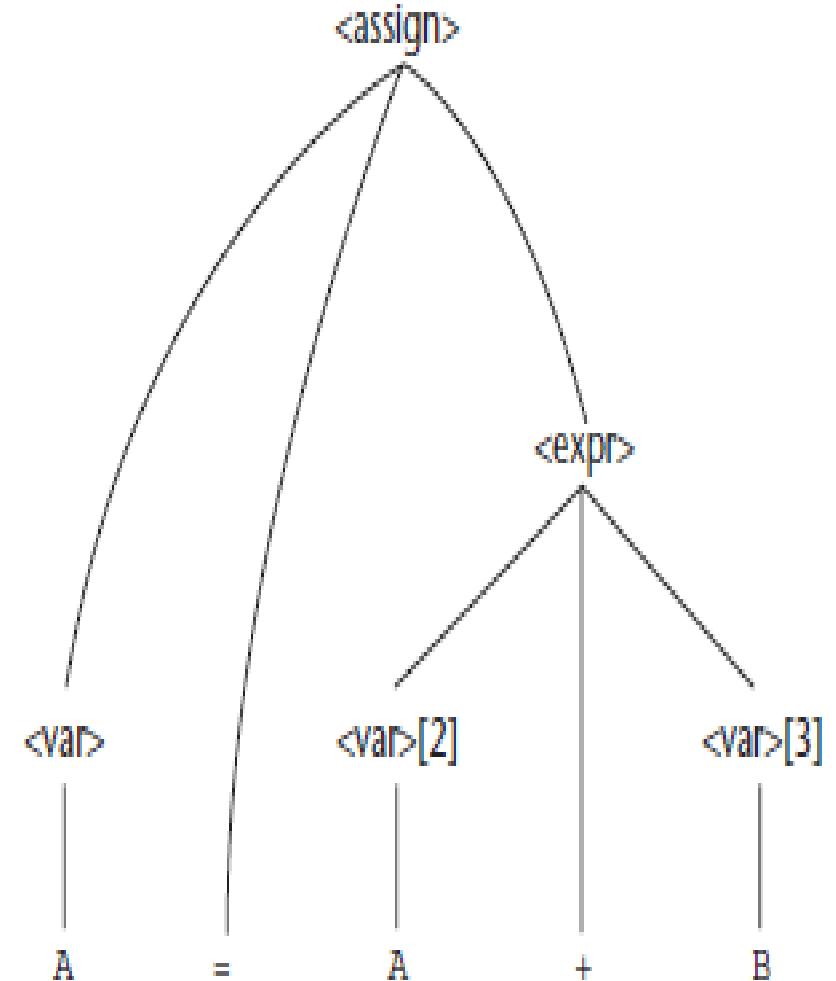
<var> → A | B

<assign> → <var> = <expr>

<expr> → <var> + <var>

<expr> → <var>

<var> → A | B



Miscellaneous

$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle$

$\color{red}{\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle}$

$\langle \text{var} \rangle \rightarrow A \mid B$

Case 1:

real A = 10.5;
int B = 6;
A = A + B;

Case 2:

int A = 10.5;
real B = 6;
A = A + B;

- Syntax and static semantics of this assignment statement are as follows:
 - Only variable names are A and B
 - Right side of the assignments can be either a variable or an expression in the form of a variable added to another variable
 - Variables can be one of two types: int or real
 - When there are two variables on the right side of an assignment, they need not be the same type
 - ✓ The type of the expression when the operand types are not the same is always real
 - ✓ When they are the same, the expression type is that of the operands
 - Type of the left side of the assignment must match the type of the right side
 - Types of operands in the right side can be mixed, but the assignment is valid only if the target and the value resulting from evaluating the right side have the same type
- The attribute grammar specifies these static semantic rules

Miscellaneous

A = A + B

$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B$

Objective: Perform Type Matching

Non-Terminals:

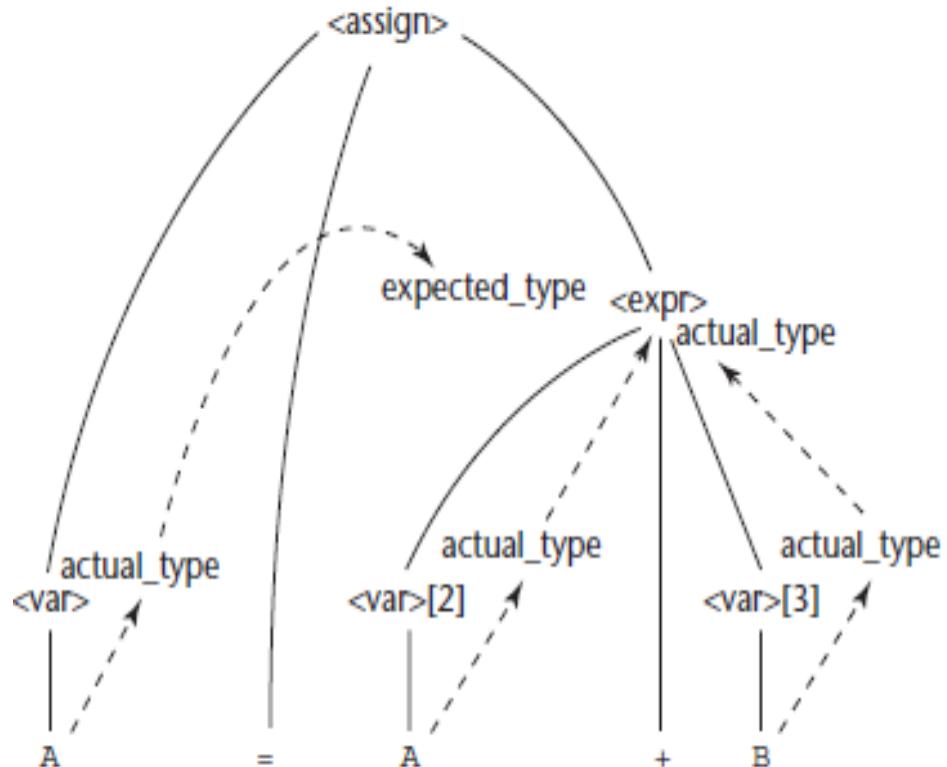
- $\langle \text{var} \rangle$
- $\langle \text{expr} \rangle$

Attributes:

- actual_type
- expected_type

Non-Terminals:

- $\langle \text{var} \rangle \rightarrow \text{actual_type}$
- $\langle \text{expr} \rangle \rightarrow \text{actual_type}, \text{expected_type}$





Actual Vs Expected Type

- *actual_type*—A synthesized attribute associated with the nonterminals <var> and <expr>. It is used to store the actual type, int or real, of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the <expr> nonterminal.
- *expected_type*—An inherited attribute associated with the nonterminal <expr>. It is used to store the type, either int or real, that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.



Actual Vs Expected Type

S.NOSynthesized Attributes

1. An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.

Inherited Attributes

1. An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.

2. The production must have non-terminal as its head.

2. The production must have non-terminal as a symbol in its body.

3. A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.

3. A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.

4. It can be evaluated during a single bottom-up traversal of parse tree.

4. It can be evaluated during a single top-down and sideways traversal of parse tree.

5. Synthesized attributes can be contained by both the terminals or non-terminals.

5. Inherited attributes can't be contained by both, It is only contained by non-terminals.

6. Synthesized attribute is used by both S-attributed SDT and L-attributed STD.

6. Inherited attribute is used by only L-attributed SDT.

EX:-
 $E.val \rightarrow F.val$



EX:-
 $E.val = F.val$



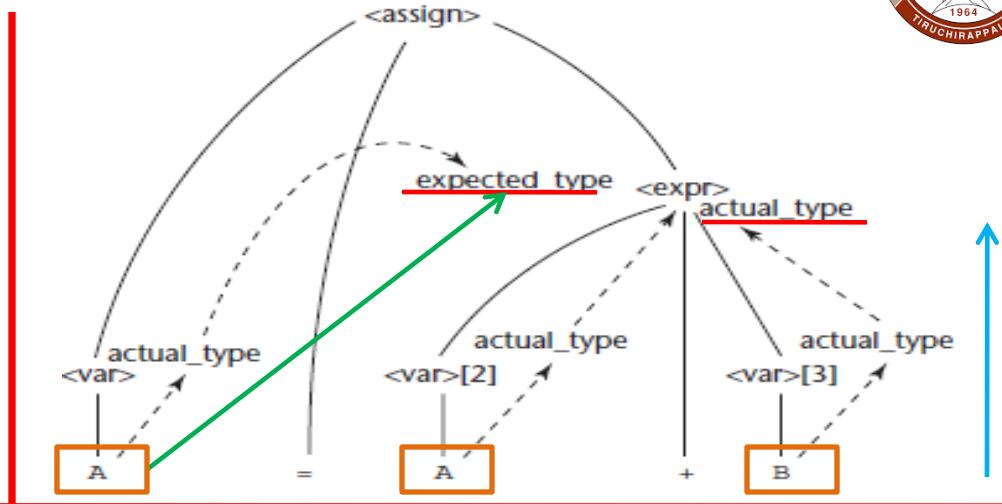
Miscellaneous

$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B$



- Syntax rule: $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

Semantic rule: $\langle \text{expr} \rangle.\text{expected_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$

- Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[2] + \langle \text{var} \rangle[3]$

Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow$

```

      if ( $\langle \text{var} \rangle[2].\text{actual\_type} = \text{int}$ ) and
          ( $\langle \text{var} \rangle[3].\text{actual\_type} = \text{int}$ )
      then int
      else real
      end if
    
```

Variable_Name	Type
A	Real
B	Int

Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$

- Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$

Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$

Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$

- Syntax rule: $\langle \text{var} \rangle \rightarrow A \mid B$

Semantic rule: $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{look-up}(\langle \text{var} \rangle.\text{string})$

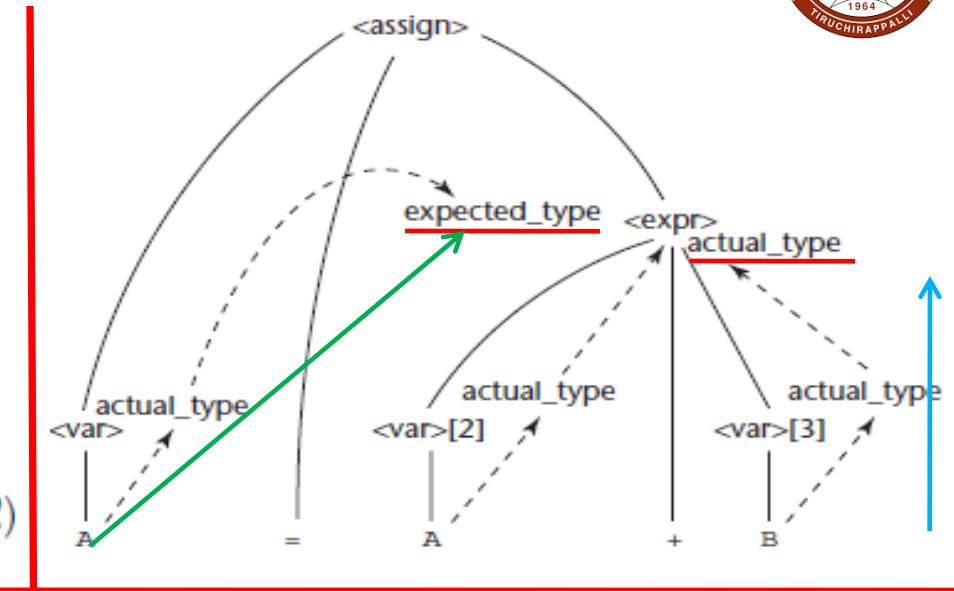


Computing Attribute Values

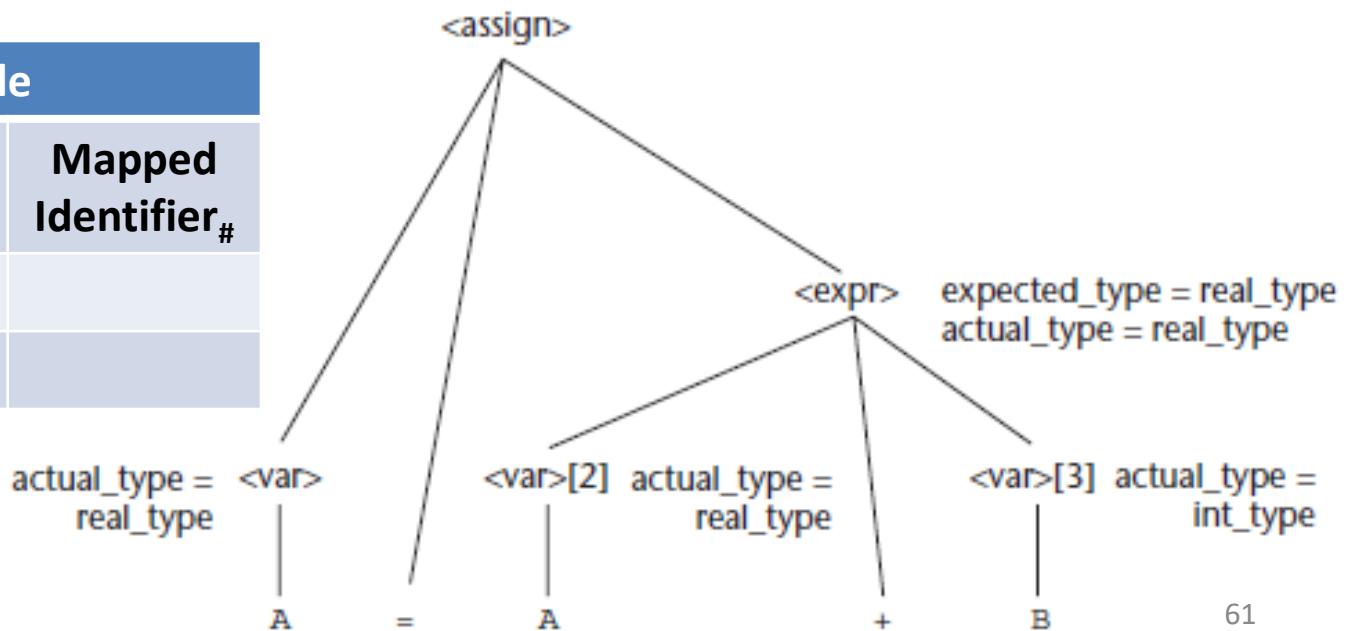
- Consider the process of computing the attribute values of a parse tree, which is sometimes called decorating the parse tree
- If all attributes were inherited, this could proceed in a completely top-down order, from the root to the leaves
- Alternatively, it could proceed in a completely bottom-up order, from the leaves to the root, if all the attributes were synthesized
- Because our grammar has both synthesized and inherited attributes, the evaluation process cannot be in any single direction

Miscellaneous

1. $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{look-up}(A)$ (Rule 4)
2. $\langle \text{expr} \rangle.\text{expected_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$ (Rule 1)
3. $\langle \text{var} \rangle[2].\text{actual_type} \leftarrow \text{look-up}(A)$ (Rule 4)
 $\langle \text{var} \rangle[3].\text{actual_type} \leftarrow \text{look-up}(B)$ (Rule 4)
4. $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \text{either int or real}$ (Rule 2)
5. $\langle \text{expr} \rangle.\text{expected_type} == \langle \text{expr} \rangle.\text{actual_type}$ is either
 TRUE or FALSE (Rule 2)



Symbol Table		
Variable Name	Data Type	Mapped Identifier #
A	real	
B	int	





Attribute Grammars

- Extension to context-free grammar
- Allows certain rules to be described conveniently → type compatibility

int a = 10.5;

- Won't raise error in lexical and syntax analysis phase
- Should raise a semantic error if the type of the assignment differs

Attribute Grammars

int a = 10.5;

- Lexical Analyzer (*Performs 3 Activities*)

Do not use this method

Lexeme	Token
int	Keyword
a	Identifier
=	Assignment Operator
10.5	Constant
;	Semi-colon

Symbol Table		
Variable Name	Data Type	Mapped Identifier _#
a	int	id ₁

id₁ = 10.5

- Syntax Analyzer

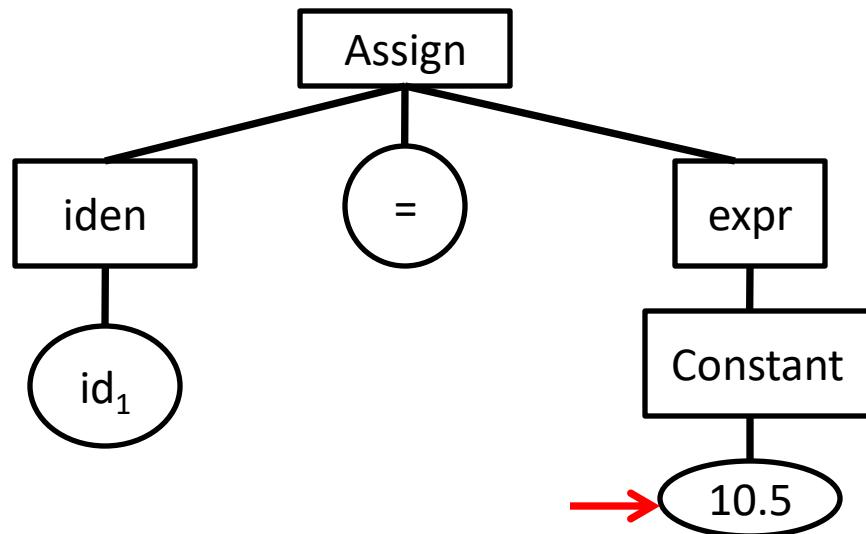
Grammar:

```

<assign> → <iden> = <expr>
<expr> → <iden> + <iden>
          | <constant>
<iden> → id1 | id2 | id3
<constant> → 10.5
    
```

Symbol Table

Variable Name	Data Type	Mapped Identifier _#
a	int	id ₁





Attribute Grammars

int a = 10.5;

- Lexical Analyzer (*Performs 3 Activities*)

Use this method

Lexeme	Token
int	Keyword
a	Identifier
=	Assignment Operator
10.5	Constant
;	Semi-colon

Symbol Table	
Variable Name	Data Type
a	int

a = 10.5

- Syntax Analyzer

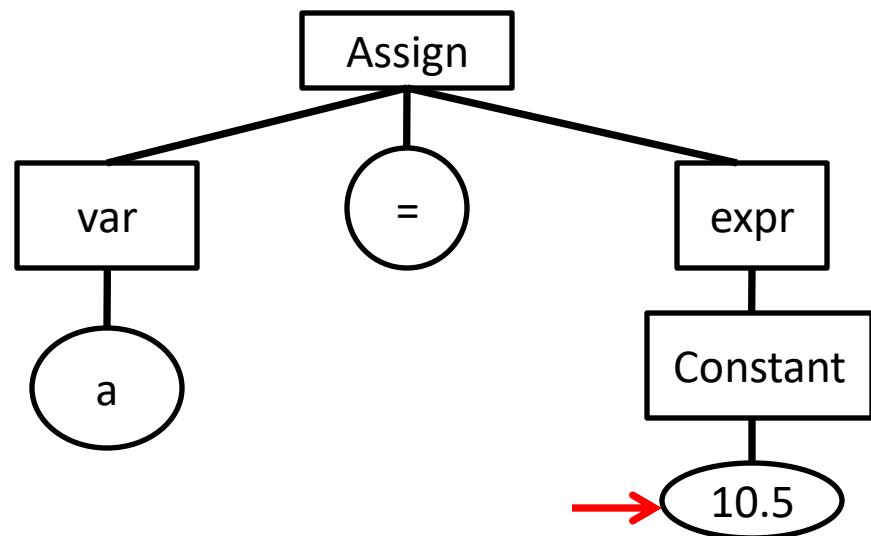
Grammar:

```

<assign> → <var> = <expr>
<expr> → <var> + <var>
          | <var>
          | <constant>
<var> → a | b
<constant> → 10.5
    
```

Symbol Table

Variable Name	Data Type
a	int





Static Semantics

- Some characteristics of programming language are difficult to describe and some are even impossible
- **Eg:** Java → Float value cannot be assigned to an int variable
 - BNF -> Requires additional non-terminal symbols and rules
 - Grammar will be too large to be useful
- **Eg:** All variables must be declared before they are referenced
 - Cannot be specified in BNF
- Introduces the categories of language rules -> Static Semantic Rules
- Static semantics of a language is only indirectly related to the meaning of programs during execution



Static Semantics

- Many static semantic rules of a language state its type constraints
- Static semantics is so named because the analysis required to check these specifications can be done at compile time
- Many powerful mechanisms have been devised to describe static semantics
 - Attribute Grammars -> Describe both syntax and static semantics of programs
- Attribute Grammars -> Formal approach to both describing and checking the correctness of static semantic rules of a program
- Although they are not always used in a formal way in compiler design, the basic concepts of attribute grammars are at least informally used in every compiler
- Dynamic Semantics -> Meaning of expressions, statements and program units



Basic Concepts

- Attribute grammars are context-free grammars with:
 - Attributes
 - Attribute Computation Functions
 - Predicate Functions
- Attributes*
 - Associated with grammar symbols (Terminals and non-terminals)
 - Similar to variables -> Can have values assigned to them
- Attribute Computation Functions*
 - Also called semantic functions
 - Associated with grammar rules
 - Used to specify how attribute values are computed
- Predicate Functions*
 - Associated with grammar rules
 - State the static semantic rules of the language

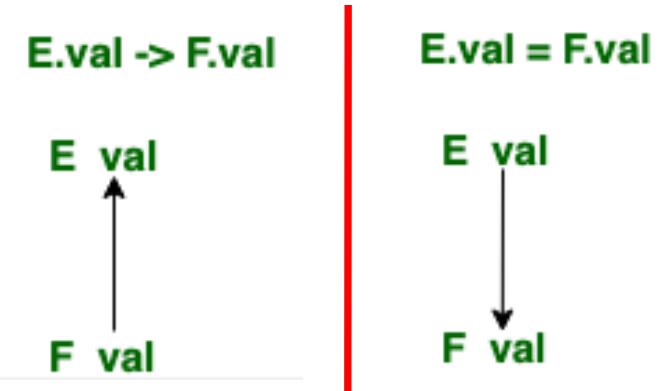
$$\begin{aligned} \langle \text{assign} \rangle &\rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle \\ \langle \text{id} \rangle &\rightarrow A \mid B \mid C \\ \langle \text{expr} \rangle &\rightarrow \langle \text{id} \rangle + \langle \text{expr} \rangle \\ &\quad \mid \langle \text{id} \rangle * \langle \text{expr} \rangle \\ &\quad \mid (\langle \text{expr} \rangle) \\ &\quad \mid \langle \text{id} \rangle \end{aligned}$$

Attribute Grammars Defined

- Grammar with the following additional features

Feature 1: Associated with each grammar symbol X is a set of attributes A(X)

- A(X) consists of two disjoint sets
 - $S(X) \rightarrow$ Synthesized Attributes
 - $I(X) \rightarrow$ Inherited Attributes
 - Synthesized attributes \rightarrow Pass semantic information up a parse tree
 - Inherited Attributes \rightarrow Pass semantic information down and across a tree





Attribute Grammars Defined

Feature 2: Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of symbols in the grammar

- For a rule, $X_0 \rightarrow X_1 \dots X_n$, the synthesized attributes of X_0 are computed with semantic functions of the form $S(X_0) = f(A(X_1), \dots, A(X_n))$
 - Value of synthesized attribute on a parse tree node depends only on the values of the attributes on that node's children nodes



Attribute Grammars Defined

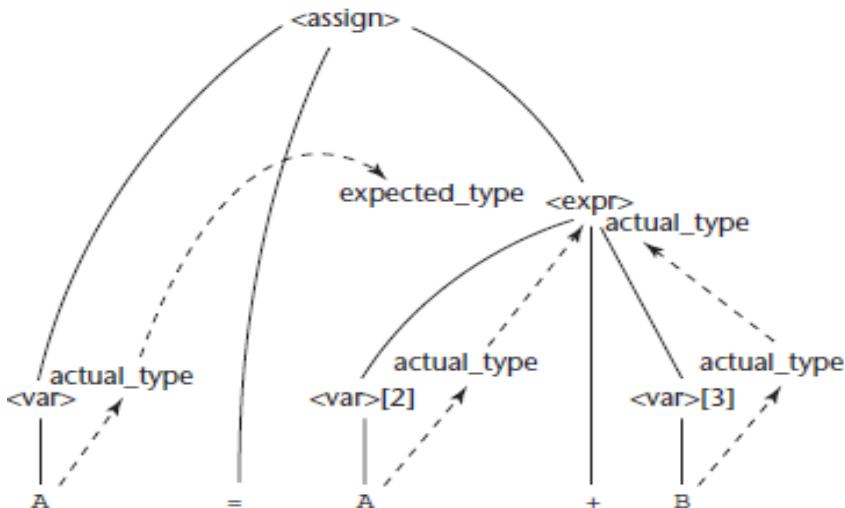
- Inherited attributes of symbols X_j , $1 \leq j \leq n$, are computed with the semantic function of the form $I(X_j) = f(A(X_0), \dots, A(X_n))$
 - Value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes
- To avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0), \dots, A(X_{j-1}))$
 - Avoids self-dependency and dependency on attributes to the right in the parse tree

Feature 3:

- Predicate function has the form of a Boolean expression on the union on the attribute set $\{A(X_0), \dots, A(X_n)\}$ and a set of literal attribute values
- Each predicate associated with every non-terminal is true

Attribute Grammars Defined

- Parse tree of an attribute grammar is the parse tree with a possibly empty set of attribute values attached to each node



- Fully attributed -> All the attribute values in the parse tree are computed



Intrinsic Attributes

- Synthesized attributes of leaf nodes whose values are determined outside the parse tree
 - Eg: Type of an instance of a variable in a program could come from the symbol table, which is used to store variable names and their types
 - Contents of the symbol table are set based on earlier declaration statements
- Initially, assuming that an unattributed parse tree has been constructed and that attribute values are needed, the only attributes with values are the intrinsic attributes of the leaf nodes
- Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values



Dynamic Semantics

C Statement

```
for (expr1; expr2; expr3) {  
    ...  
}
```

Meaning

```
expr1;  
loop: if expr2 == 0 goto out  
      ...  
      expr3;  
      goto loop  
out:   ...
```



Describing the Meanings of Programs: Dynamic Semantics

- Dynamic Semantics -> Meaning of expressions, statements and program units of a programming language
 - Programmars obviously need to know precisely what statements of a language do before they can use them effectively in their programs
 - Compiler writers must know exactly what language constructs mean to design implementations for them correctly
- If there were a precise semantics specification of a programming language
 - Programs written in the language potentially could be proven correct without testing
 - Compilers could be shown to produce programs that exhibited exactly the behavior given in the language definition; that is, their correctness could be verified
 - A complete specification of the syntax and semantics of a programming language could be used by a tool to generate a compiler for the language automatically
 - Finally, language designers, who would develop the semantic descriptions of their languages, could in the process discover ambiguities and inconsistencies in their designs



Describing the Meanings of Programs: Dynamic Semantics

- Software developers and compiler designers typically determine the semantics of programming languages by reading English explanations in language manuals
 - Because such explanations are often imprecise and incomplete, this approach is clearly unsatisfactory
- Due to the lack of complete semantics specifications of programming languages, programs are rarely proven correct without testing, and commercial compilers are never generated automatically from language descriptions



Operational Semantics

- Idea behind operational semantics is to describe the meaning of a statement or program by specifying the effects of running it on a machine
 - Effects on the machine are viewed as the sequence of changes in its state
 - Machine's state is the collection of the values in its storage
- An obvious operational semantics description, then, is given by executing a compiled version of the program on a computer
- Several problems with using this approach for complete formal semantics descriptions
 - First, the individual steps in the execution of machine language and the resulting changes to the state of the machine are too small and too numerous
 - Second, the storage of a real computer is too large and complex -> There are usually several levels of memory devices, as well as connections to enumerable other computers and memory devices through networks
 - Machine languages and real computers are not used for formal operational semantics
- Intermediate-level languages and interpreters for idealized computers are designed specifically for the process



Operational Semantics

- There are different levels of uses of operational semantics
 - At the highest level, the interest is in the final result of the execution of a complete program -> Natural Operational Semantics
 - At the lowest level, operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed -> Structural Operational Semantics



Basic Process

- First step in creating an operational semantics description of a language is to design an appropriate intermediate language
 - Primary focus is clarity
- Every construct of intermediate language must have an unambiguous meaning
- If the semantics description is to be used for natural operational semantics, a virtual machine (an interpreter) must be constructed for the intermediate language
 - Virtual machine can be used to execute either single statements, code segments, or whole programs
 - Semantics description can be used without a virtual machine if the meaning of a single statement is all that is required
 - In this use, which is structural operational semantics, the intermediate code can be visually inspected



Basic Process

- Basic process of operational semantics is not unusual -> Concept is frequently used in programming textbooks and programming language reference manuals

C Statement

```
for (expr1; expr2; expr3) {  
    ...  
}
```

Meaning

```
expr1;  
loop: if expr2 == 0 goto out  
    ...  
    expr3;  
    goto loop  
out:   ...
```

- Human reader of such a description is the virtual computer and is assumed to be able to “execute” the instructions in the definition correctly and recognize the effects of the “execution”
- Intermediate language and its associated virtual machine used for formal operational semantics descriptions are often highly abstract
- Intermediate language is meant to be convenient for the virtual machine, rather than for human readers



Basic Process

- For our purposes, however, a more human-oriented intermediate language could be used

```
ident = var  
ident = ident + 1  
ident = ident - 1  
goto label  
if var relop var goto label
```

ident -> Identifier
var -> Identifier or Constant
relop -> {=, <>, >, <, >=, <=}

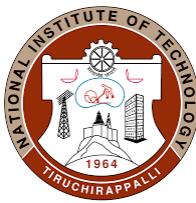
- Statements are all simple and therefore easy to understand and implement
- Slight generalization of assignment statements allows more general arithmetic expressions and assignment statements to be described

```
ident = var bin_op var  
ident = un_op var
```

- Multiple arithmetic data types and automatic type conversions, of course, complicate this generalization
- Adding just a few more relatively simple instructions would allow the semantics of arrays, records, pointers, and subprograms to be described



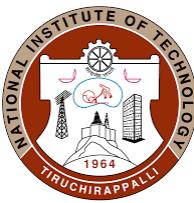
Thank You



CSPC31: Principles of Programming Languages

Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 4 – Lexical and Syntax Analysis



Need for this Chapter

- Syntax analyzers rely on the grammars
 - Application of grammars
- Lexical and Syntax analyzers are needed in numerous situations outside compiler design
 - Compute the complexity of programs
 - Analyze and react to the contents of a configuration file



Objectives

- Introduction to Lexical Analysis
- Discusses the general parsing problem
- Recursive-descent implementation technique -> Top-down parsers
- Bottom-up Parsing -> LR Parsing Algorithm



Introduction

- Three different approaches to implement programming languages
 - Compilation, Pure Interpretation, Hybrid
- Compilation -> C++ and COBOL
- Pure Interpretation -> used for smaller systems in which execution efficiency is not critical. **Eg:** JavaScript
- Hybrid -> Java, Perl (JIT improves speed)
- Syntax analyzers are nearly always based on a formal description of the syntax of programs
- Most commonly used syntax-description formalism is context-free grammars or BNF



Introduction

- Reasons
 - Clear and concise
 - Can be used as the direct basis for syntax analyzer
 - Implementations are relatively easy
- Syntax Analyzing Task -> Lexical Analysis + Syntax Analysis
- Lexical Analysis -> Small-scale language constructs (Names, Numeric Literals)
- Syntax Analysis -> Large-scale constructs (Expressions, Statements and Program Units)
- Reasons for separation
 - Simplicity
 - Efficiency
 - Portability



Lexical Analysis

- Pattern matcher
- Serves as a front end of a syntax analyzer
- Collects characters into logical groupings (lexemes) and assigns internal codes to the groupings (tokens)

result = oldsum – value / 100;

<u>Token</u>	<u>Lexeme</u>
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;



Lexical Analysis

- Role of Lexical Analyzer
 - Skips comments and white space outside lexemes
 - Inserts lexemes for user-defined names into the symbol table
 - Detects syntactic errors in tokens
- Possible ways to build
 - Write a formal description of the token patterns (lex)
 - State transition diagram + Program
 - State transition diagram + Table-driven implementation



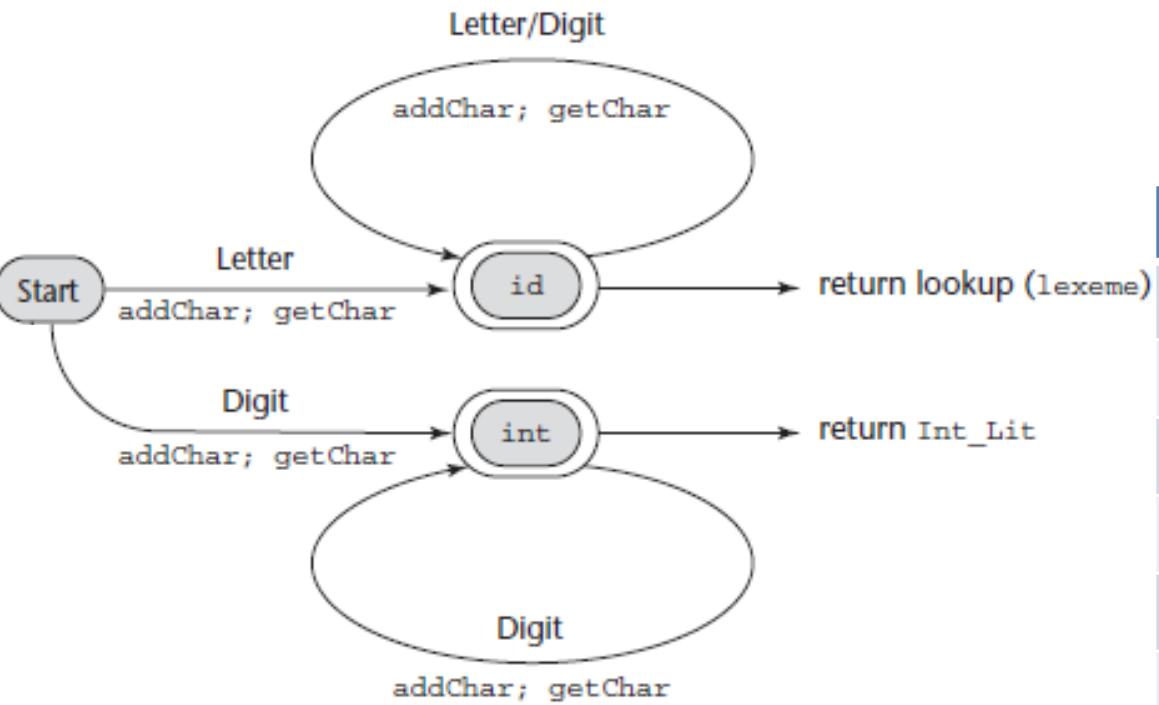
Lexical Analysis

- State Transition Diagram
 - Directed graph
 - Nodes -> State names
 - Arcs -> Labelled with input characters that cause the transitions
- State diagrams used for lexical analyzers are representations of a class of mathematical machines called finite automata
- Finite automata -> Used to recognize members of a class of languages called regular languages
- Tokens of a programming language -> Regular language
- Lexical analyzer is a finite automaton

Lexical Analysis

- Recognize program names, reserved words and int literals
- Names -> Uppercase letters, Lowercase letters and Digits
 - Must begin with a letter, No length limitation

**int a;
float b;**



Name	Type
a	int
b	float

Token	Lexeme	Int
Keyword	int	Float
Identifier	a	If
Semi-colon	;	Else
Keyword	float	Char
Identifier	b	.
Semi-colon	;	.

Finite Automaton

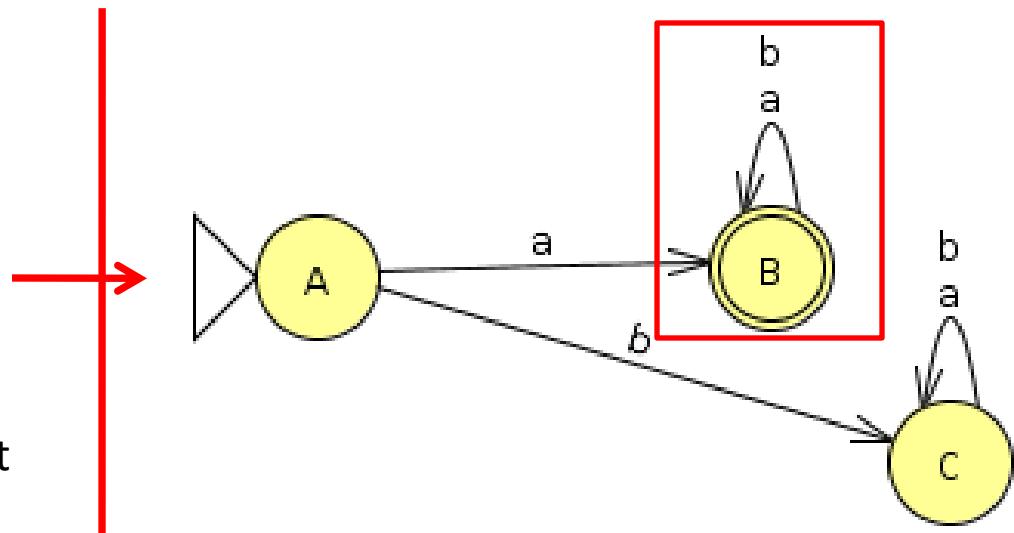
Eg 1: $aabbbaaabbbbabababababa$

$\Rightarrow a.(a \mid b)^+$ **(Accepted)**

Eg 2: $babababbbbabababaaba$

$\Rightarrow b.(a \mid b)^+$ **(Rejected)**

This accepts all strings over a, b that start with an 'a'.



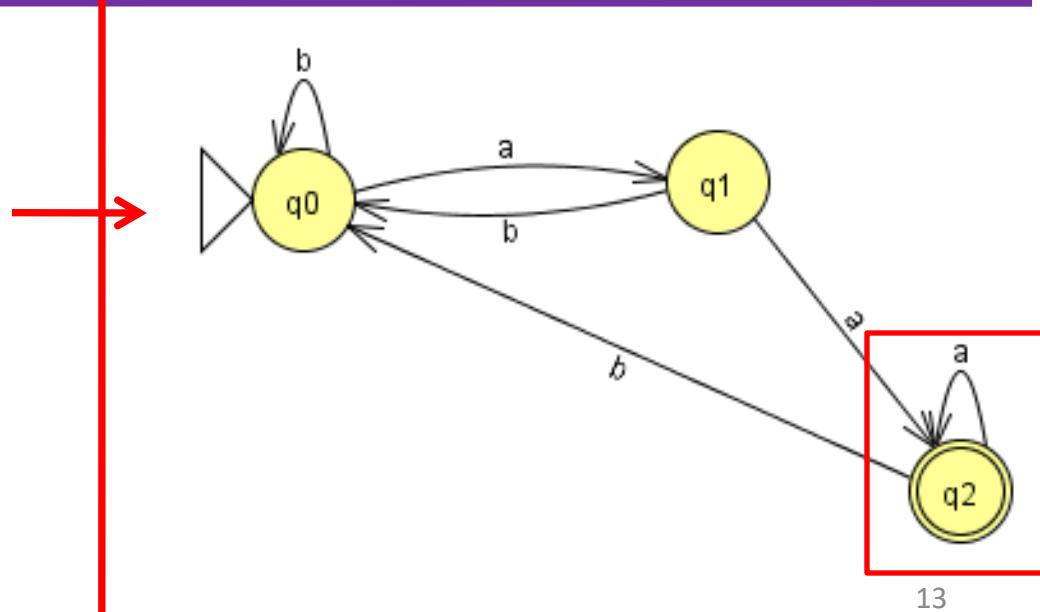
Eg 1: $bbbbbabbbbaaaaaaa$

$\Rightarrow b^+.(a \mid b)^+.a.a$ **(Accepted)**

Eg 2: $bbbbbabbbbaaaaaaaaabb$

$\Rightarrow b^+.(a \mid b)^+.b.b$ **(Rejected)**

This accepts all strings over a, b that end with two a's.





Lexical Analysis

- Responsible for the initial construction of symbol table
- User-defined names + Attributes

```
float a;  
int b;
```

Variable_Name	Type
a	float
b	int



Parsing Problem / Parsing Decision Problem

- Part of the process of analyzing syntax -> Syntax Analysis or Parsing
- Types
 - Top-down Parsing
 - Bottom-up Parsing



Introduction to Parsing

- Goals
 - Must check the input program to determine whether it is syntactically correct
 - Produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input
- Two categories
 - Top-down
 - Bottom-up



Introduction to Parsing

- **Terminal symbols -> Lowercase letters** at the beginning of the alphabet (**a, b, . . .**)
- **Non-terminal symbols -> Uppercase letters** at the beginning of the alphabet (**A, B, . . .**)
- Terminals or non-terminals -> Uppercase letters at the end of the alphabet (**W, X, Y, Z**)
- **Strings of terminals -> Lowercase letters** at the end of the alphabet (**w, x, y, z**)
- **Mixed strings (terminals and/or non-terminals) -> Lowercase Greek letters (α, β, γ, δ)**



Introduction to Parsing

- For programming languages, terminal symbols are the small-scale syntactic constructs of the language, what we have referred to as lexemes

Eg: int, float, a, b

- Nonterminal symbols of programming languages are usually connotative names or abbreviations, surrounded by pointed brackets

Eg: <expr>, <A>

- Sentences of a language (programs, in the case of a programming language) are strings of terminals

Eg: if x == 5 then x = x +1 else x = x - 1, a + b

- Mixed strings describe right-hand sides (RHSs) of grammar rules and are used in parsing algorithms

Eg: if <logic_expr> then <stmt> else <stmt>



Top-Down Parsers

- Builds the parse tree in pre-order -> Leftmost Derivation
- Given a sentential form -> Finds the next sentential form in that leftmost derivation
- General notation -> $xA\alpha$
 - x -> String of Terminal Symbols
 - A -> Non-terminal
 - α -> Mixed-string
- Determining the next sentential form is a matter of choosing the correct grammar rule that has A as its LHS
 - A-rules: $A \rightarrow bB \mid cBb \mid a$

if $x==5$ then $x = x + 1$ else $x = x - 1$

if <logic_expr> then <stmt> else <stmt>
x A α

logic_expr -> var == literal
| var >= literal



Top-Down Parsers

- General notation $\rightarrow xAa$
 - A-rules: $A \rightarrow bB \mid cBb \mid a$
- Different top-down parsing algorithms use different information to make parsing decisions
 - Most common top-down parsers choose the correct RHS for the leftmost nonterminal in the current sentential form by comparing the next token of input with the first symbols that can be generated by the RHSs of those rules
 - Whichever RHS has that token at the left end of the string it generates, is the correct one
- In the sentential form xA , the parser would use whatever token would be the first generated by A to determine which A-rule should be used to get the next sentential form
 - In the example, the three RHSs of the A-rules all begin with different terminal symbols
 - The parser can easily choose the correct RHS based on the next token of input, which must be a, b, or c in this example
- In general, choosing the correct RHS is not so straightforward, because some of the RHSs of the leftmost nonterminal in the current sentential form may begin with a nonterminal
- Types
 - Recursive-Descent Parser \rightarrow Coded version
 - Parsing Table
- LL Algorithm \rightarrow **Left-to-Right Scan, Leftmost Derivation**

Miscellaneous

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \underline{\langle \text{expr} \rangle + \langle \text{term} \rangle}$
 $\quad \mid \langle \text{term} \rangle$

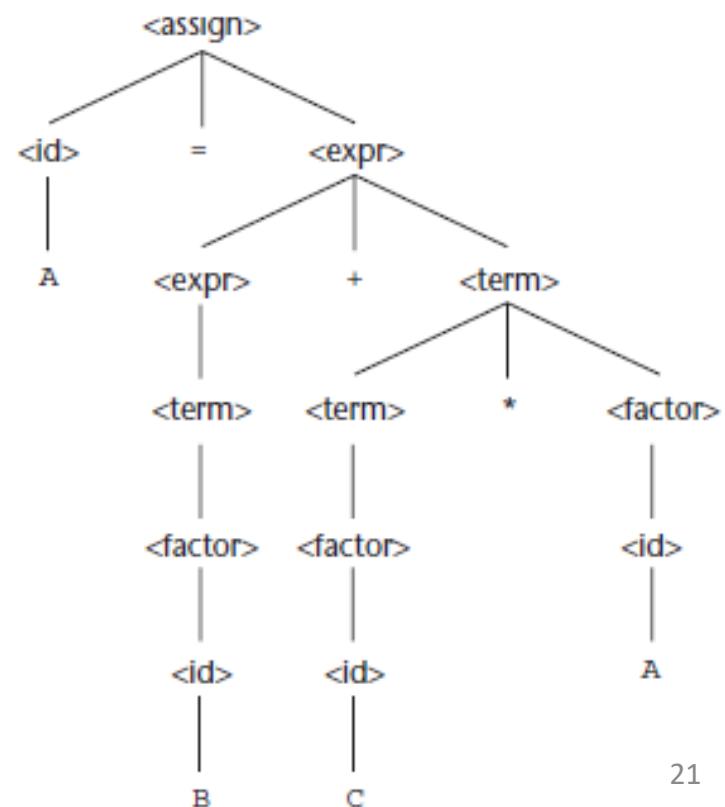
$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\quad \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$
 $\quad \mid \langle \text{id} \rangle$

$$A = B + C * A$$

Leftmost Derivation

$\langle \text{assign} \rangle \Rightarrow \underline{\langle \text{id} \rangle} = \langle \text{expr} \rangle$
 $\Rightarrow A = \underline{\langle \text{expr} \rangle}$
 $\Rightarrow A = \underline{\langle \text{expr} \rangle} + \langle \text{term} \rangle$
 $\Rightarrow A = \underline{\langle \text{term} \rangle} + \langle \text{term} \rangle$
 $\Rightarrow A = \underline{\langle \text{factor} \rangle} + \langle \text{term} \rangle$
 $\Rightarrow A = \underline{\langle \text{id} \rangle} + \langle \text{term} \rangle$
 $\Rightarrow A = B + \langle \text{term} \rangle$
 $\Rightarrow A = B + \underline{\langle \text{term} \rangle * \langle \text{factor} \rangle}$
 $\Rightarrow A = B + \underline{\langle \text{factor} \rangle * \langle \text{factor} \rangle}$
 $\Rightarrow A = B + \underline{\langle \text{id} \rangle * \langle \text{factor} \rangle}$
 $\Rightarrow A = B + C * \underline{\langle \text{factor} \rangle}$
 $\Rightarrow A = B + C * \underline{\langle \text{id} \rangle}$
 $\Rightarrow A = B + C * A$





Bottom-Up Parsers

- Constructs a parse tree by beginning at the leaves and progressing toward the root
 - Reverse of the rightmost derivation
- Sentential forms of the derivation are produced in order of last to first
- In terms of the derivation, a bottom-up parser can be described as follows:
 - Given a right sentential form α , the parser must determine what substring of α is the RHS of the rule in the grammar that must be reduced to its LHS to produce the previous sentential form in the rightmost derivation
 - Eg: First step for a bottom-up parser is to determine which substring of the initial given sentence is the RHS to be reduced to its corresponding LHS to get the second last sentential form in the derivation

Input: aabc

$$\begin{array}{l} S \rightarrow aAc \\ A \rightarrow aA \mid b \end{array}$$

$S \Rightarrow aAc \Rightarrow aaAc \Rightarrow aabc$

$S \rightarrow aabc$
 $S \rightarrow aaAc \leftarrow$
 $S \rightarrow aAc$



Bottom-Up Parsers

- Process of finding the correct RHS to reduce is complicated by the fact that a given right sentential form may include more than one RHS from the grammar of the language being parsed
- Correct RHS is called the **handle**



Top-Down Parser

Recursive-Descent Parsing

- Consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order
- EBNF is ideally suited for recursive-descent parsers
 - {} -> 0 or more times
 - [] -> once or not at all

`<if_statement> → if <logic_expr> <statement> [else <statement>]
<ident_list> → ident [, ident]`



Recursive-Descent Parsing

- A recursive-descent parser has a subprogram for each non-terminal in its associated grammar
- When given an input string, it traces out the parse tree that can be rooted at that non-terminal and whose leaves match the input string
- A recursive-descent parsing subprogram is a parser for the language (set of strings) that is generated by its associated non-terminal

$$\begin{aligned}\underline{\text{<expr>} \rightarrow \text{<term>} \{ (+ \mid -) \text{ <term>} \}} \\ \underline{\text{<term>} \rightarrow \text{<factor>} \{ (* \mid /) \text{ <factor>} \}} \\ \underline{\text{<factor>} \rightarrow \text{id} \mid \text{int_constant} \mid (\text{ <expr>})}\end{aligned}$$



Recursive Descent Parsing

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

```
/* expr  
   Parses strings in the language generated by the rule:  
   <expr> -> <term> { (+ | -) <term>}  
 */  
void expr() {  
    printf("Enter <expr>\n");  
  
    /* Parse the first term */  
    term();  
  
    /* As long as the next token is + or -, get  
       the next token and parse the next term */  
    while (nextToken == ADD_OP || nextToken == SUB_OP) {  
        lex();  
        term();  
    }  
    printf("Exit <expr>\n");  
} /* End of function expr */
```

a + b

Does not include code for syntax
error detection or recovery



Recursive Descent Parsing

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle\} \\ \langle \text{term} \rangle &\rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle\} \\ \langle \text{factor} \rangle &\rightarrow \text{id} \mid \text{int_constant} \mid (\langle \text{expr} \rangle)\end{aligned}$$

```
/* term
   Parses strings in the language generated by the rule:
   <term> -> <factor> { (* | /) <factor>}
   */
void term() {
    printf("Enter <term>\n");

    /* Parse the first factor */
    factor();

    /* As long as the next token is * or /, get the
       next token and parse the next factor */
    while (nextToken == MULT_OP || nextToken == DIV_OP) {
        lex();
        factor();
    }
    printf("Exit <term>\n");
} /* End of function term */
```



Recursive Descent Parsing

```
/* factor
   Parses strings in the language generated by the rule:
   <factor> -> id | int_constant | ( <expr> )
*/
void factor() {
    printf("Enter <factor>\n");

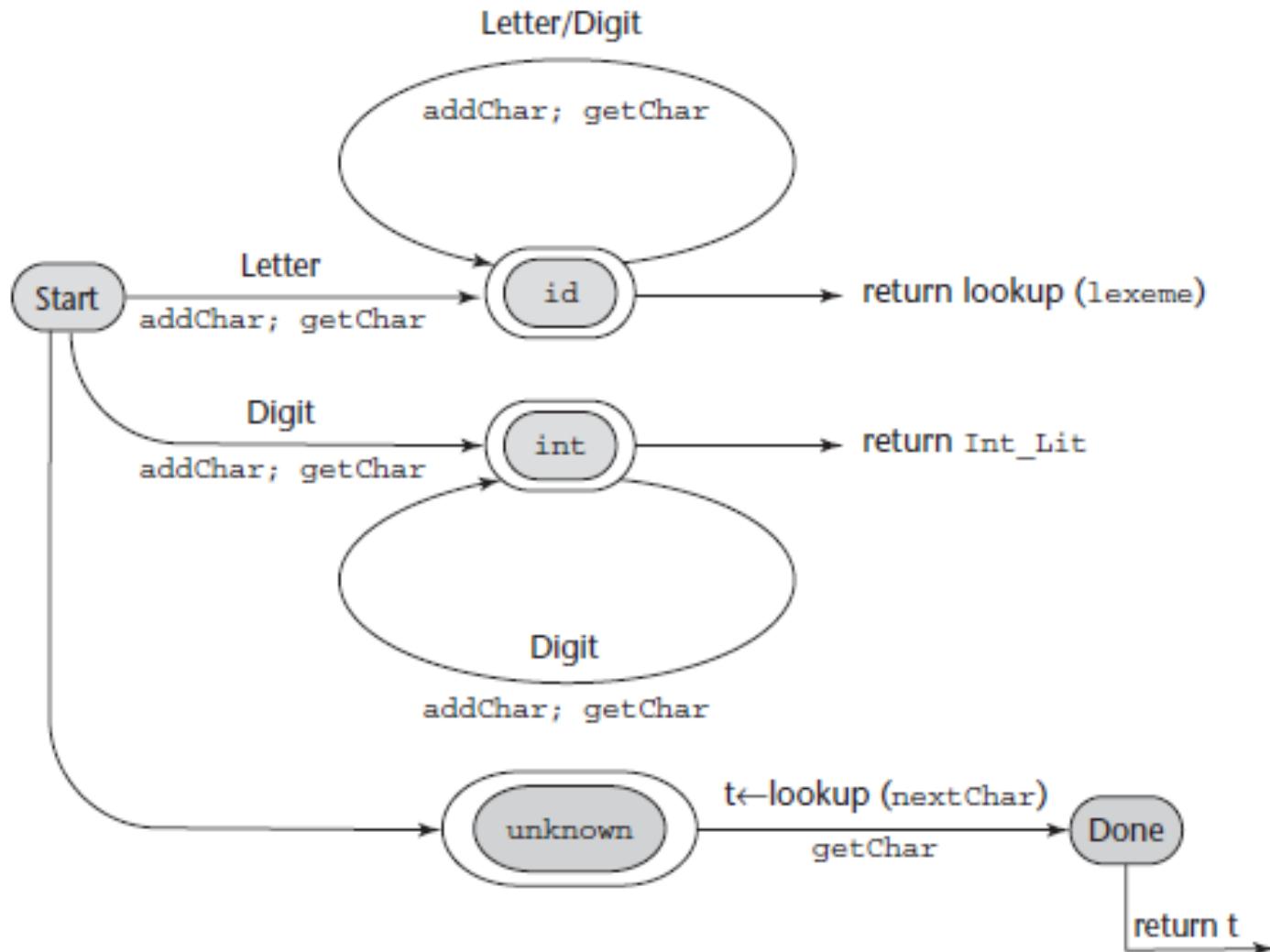
    /* Determine which RHS */
    if (nextToken == IDENT || nextToken == INT_LIT)

    /* Get the next token */
    lex();

    /* If the RHS is ( <expr> ), call lex to pass over the
       left parenthesis, call expr, and check for the right
       parenthesis */
    else {
        if (nextToken == LEFT_PAREN) {
            lex();
            expr();
            if (nextToken == RIGHT_PAREN)
                lex();
            else
                error();
        } /* End of if (nextToken == ... */

        /* It was not an id, an integer literal, or a left
           parenthesis */
        else
            error();
    } /* End of else */
}
```

Recursive Descent Parsing



Recursive Descent Parsing

(sum + 47) / total

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
```

```
Next token is: 11 Next lexeme is sum
Enter <expr>
```

```
Enter <term>
Enter <factor>
```

```
Next token is: 21 Next lexeme is +
Exit <factor>
```

```
Exit <term>
```

```
Next token is: 10 Next lexeme is 47
```

```
Enter <term>
Enter <factor>
```

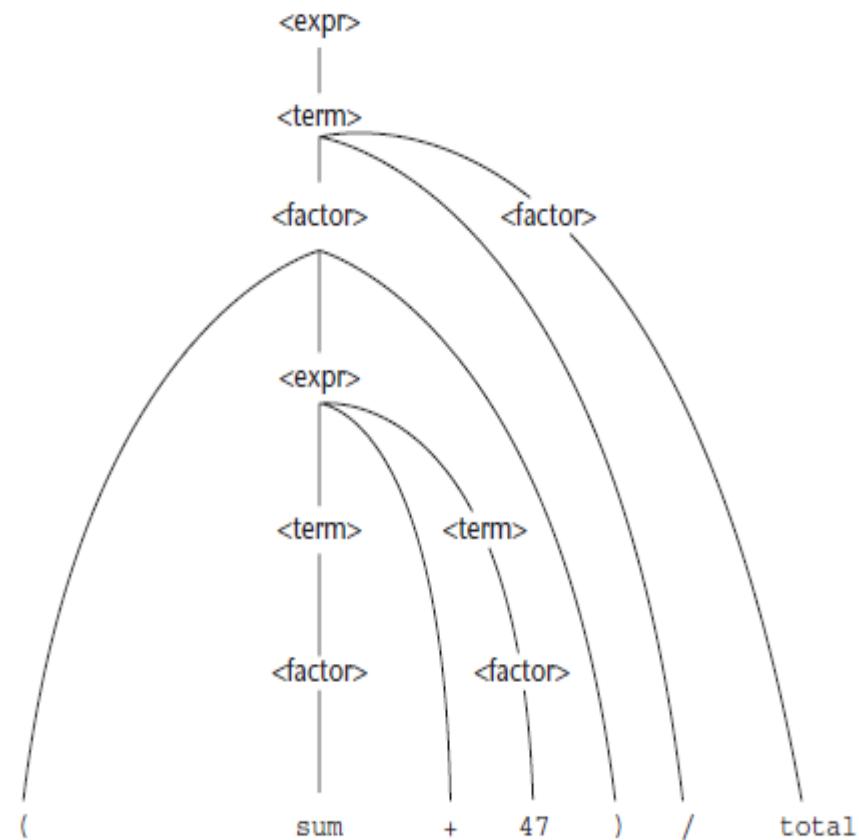
```
Next token is: 26 Next lexeme is )
Exit <factor>
```

```
Exit <term>
Exit <expr>
```

```
Next token is: 24 Next lexeme is /
Exit <factor>
```

```
Next token is: 11 Next lexeme is total
Enter <factor>
```

```
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

$$\begin{aligned} \text{<expr>} &\rightarrow \text{<term>} \{ (+ \mid -) \text{<term>} \} \\ \text{<term>} &\rightarrow \text{<factor>} \{ (\ast \mid /) \text{<factor>} \} \\ \text{<factor>} &\rightarrow \text{id} \mid \text{int_constant} \mid (\text{<expr>}) \end{aligned}$$




LL Grammar Class

- Left Recursion is a problem for LL Parsers (Recursive Descent)

$$A \rightarrow A + B$$

- A recursive-descent parser subprogram for A immediately calls itself to parse the first symbol in its RHS
- That activation of the A parser subprogram then immediately calls itself again, and again, and so forth
- **Two Types:** Direct and Indirect Left Recursion
- Direct Left Recursion

ϵ specifies the empty string. A rule that has ϵ as its RHS is called an erasure rule, because its use in a derivation effectively erases its LHS from the sentential form.

For each nonterminal, A,

1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \underline{\beta_1} \mid \beta_2 \mid \dots \mid \underline{\beta_n}$ where none of the β 's begins with A
2. Replace the original A-rules with

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$\epsilon \rightarrow$ Erasure Rule

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_m A' \mid \epsilon$$



LL Grammar Class

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

For the E-rules, we have $\alpha_1 = + T$ and $\beta = T$, so we replace the E-rules with

$$E \rightarrow T E'$$
$$E' \rightarrow + T E' \mid \epsilon$$

For the T-rules, we have $\alpha_1 = * F$ and $\beta = F$, so we replace the T-rules with

$$T \rightarrow F T'$$
$$T' \rightarrow * F T' \mid \epsilon$$



LL Grammar Class

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$
$$\Rightarrow$$
$$E \rightarrow T E'$$
$$E' \rightarrow + T E' \mid \epsilon$$
$$T \rightarrow F T'$$
$$T' \rightarrow * F T' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$



LL Grammar Class

- Indirect Left Recursion also poses same problem

$$A \rightarrow BaA$$

$$B \rightarrow Ab$$

- A recursive-descent parser for these rules would have the A subprogram immediately call the subprogram for B, which immediately calls the A subprogram
 - Indirect Left Recursion -> Solution exists (not covered here)
- Problem for all top-down parsing algorithms
- Left recursion is not a problem for bottom-up parsing algorithms



LL Grammar Class

- Left recursion is not the only grammar trait that disallows top-down parsing
- Another is whether the parser can always choose the correct RHS on the basis of the next token of input, using only the first token generated by the leftmost nonterminal in the current sentential form
- There is a relatively simple test of a non-left recursive grammar that indicates whether this can be done, called the pairwise disjointness test

$$\begin{aligned} \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\ &\quad | \quad \langle \text{term} \rangle \end{aligned}$$

- Pairwise Disjointness Set -> Which RHS to choose

$\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$ (If $\alpha \Rightarrow^* \epsilon$, ϵ is in $\text{FIRST}(\alpha)$)
in which \Rightarrow^* means 0 or more derivation steps.



LL Grammar Class

The pairwise disjointness test is as follows:

For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$$

(The intersection of the two sets, $\text{FIRST}(\alpha_i)$ and $\text{FIRST}(\alpha_j)$, must be empty.)

- If a non-terminal A has more than one RHS, the first terminal symbol that can be generated in a derivation for each of them must be unique to that RHS

$$A \rightarrow aB \mid bAb \mid Bb$$

$$B \rightarrow cB \mid d$$

- FIRST sets for the RHSs of the A-rules are {a}, {b}, {c, d}



LL Grammar Class

$$\begin{aligned} A &\rightarrow aB \mid BA_b \\ B &\rightarrow aB \mid b \end{aligned}$$

- FIRST sets for the RHSs in the A-rules are {a}, {a, b} -> Fails the test
- In many cases, a grammar that fails the pairwise disjointness test can be modified so that it will pass the test

$\langle \text{variable} \rangle \rightarrow \text{identifier} \mid \text{identifier} [\text{expression}]$

- Do not pass pairwise disjointness test
- This problem can be alleviated through a process called left factoring
- Parts that follow identifier in the two RHSs are (the empty string) and [expression]

$\langle \text{variable} \rangle \rightarrow \text{identifier} \langle \text{new} \rangle$

$\langle \text{new} \rangle \rightarrow e \mid [\text{expression}]$

- EBNF: $\langle \text{variable} \rangle \rightarrow \text{identifier} [[\text{expression}]]$
- Left factoring cannot solve all pairwise disjointness problems of grammars
- Sometimes, rules must be rewritten in other ways to eliminate the problem



Bottom-Up Parsing - Parsing Problem

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array}$$

id + id * id

- LL Algorithm -> Left-to-Right Scan, Rightmost Derivation in Reverse
- Left-Recursion -> No problem
- No metasymbols

**Rightmost
Derivation in
Reverse**

E => E + T
E + T * F
E + T * id
E + F * id
E + id * id
T + id * id
F + id * id
id + id * id



Bottom-Up Parsing - Parsing Problem

- Underlined part of each sentential form is the RHS that is rewritten as its corresponding LHS to get the previous sentential form
- Bottom-up parsing produces the reverse of a rightmost derivation
- Task is to find the specific RHS that must be rewritten to get the next (previous) sentential form

$E + T * id$

- Three RHSs, $E + T$, T , id -> Only one of these is the handle
- If $E + T$ is chosen $=> E * id$ -> Not a legal right sentential form
- Handle of a right sentential form is unique
- Task of a bottom-up parser is to find the handle of any given right sentential form that can be generated by its associated grammar

$$\begin{aligned}
 E &\rightarrow \underline{E + T} \mid T \\
 T &\rightarrow \underline{T * F} \mid F \\
 F &\rightarrow (E) \mid \underline{id}
 \end{aligned}$$

$$\begin{aligned}
 E &\Rightarrow \underline{E + T} \\
 &\Rightarrow E + \underline{T * F} \\
 &\Rightarrow E + \underline{T * \underline{id}} \\
 &\Rightarrow E + \underline{F * id} \\
 &\Rightarrow E + \underline{id * id} \\
 &\Rightarrow \underline{T + id * id} \\
 &\Rightarrow \underline{F + id * id} \\
 &\Rightarrow \underline{id + id * id}
 \end{aligned}$$

Bottom-Up Parsing - Parsing Problem

\Rightarrow^*_{rm} specifies zero or more rightmost derivation steps

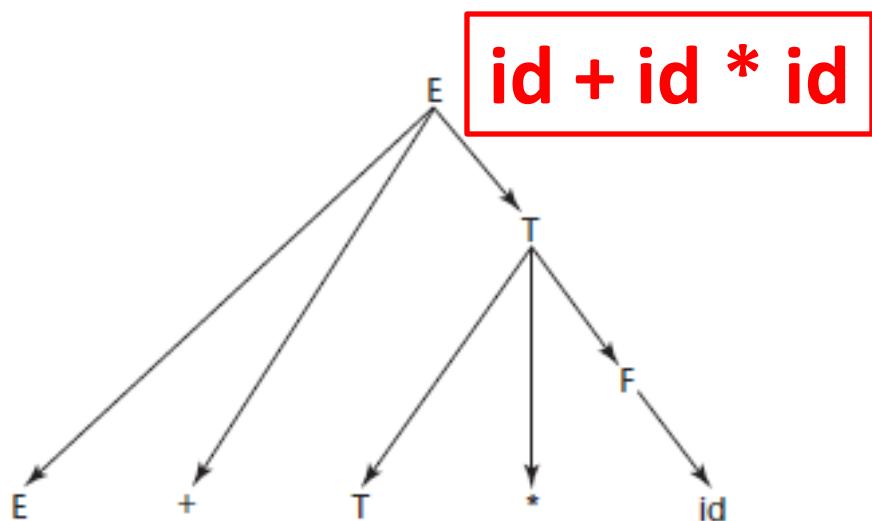
Definition: β is the handle of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

Definition: β is a phrase of the right sentential form γ if and only if

$S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^* + \alpha_1 \beta \alpha_2$

\Rightarrow^+_{rm} specifies one or more rightmost derivation steps

Definition: β is a simple phrase of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^* \alpha_1 \beta \alpha_2$



$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

Phrase:

- $E + T * id$
- $T * id$
- id

Simple Phrase:

- id



Introduction to Parsing

- **Terminal symbols -> Lowercase letters** at the beginning of the alphabet (**a, b, . . .**)
- **Non-terminal symbols -> Uppercase letters** at the beginning of the alphabet (**A, B, . . .**)
- Terminals or non-terminals -> Uppercase letters at the end of the alphabet (**W, X, Y, Z**)
- **Strings of terminals -> Lowercase letters** at the end of the alphabet (**w, x, y, z**)
- **Mixed strings (terminals and/or non-terminals) -> Lowercase Greek letters** (**$\alpha, \beta, \gamma, \delta$**)

Bottom-Up Parsing - Parsing Problem



The definitions of phrase and simple phrase may appear to have the same lack of practical value as that of a handle, but that is not true. Consider what a phrase is relative to a parse tree. It is the string of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree. A simple phrase is just a phrase that takes a single derivation step from its root nonterminal node. In terms of a parse tree, a phrase can be derived from a single nonterminal in one or more tree levels, but a simple phrase can be derived in just a single tree level. Consider the parse tree shown in Figure 4.3.

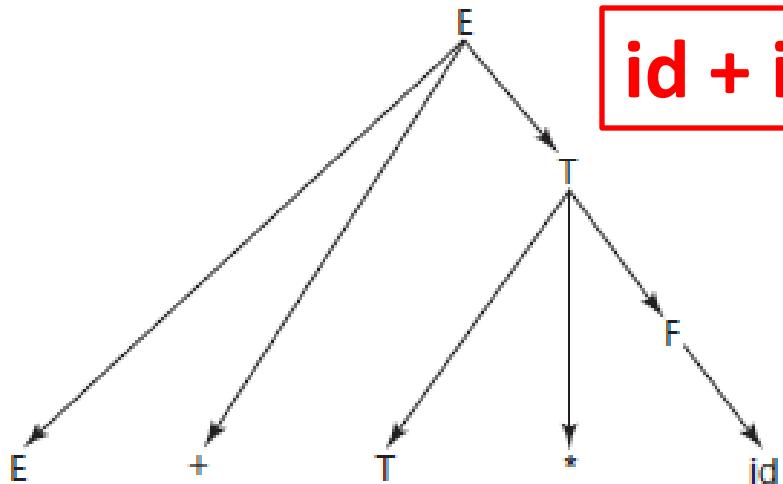
Level 0

id + id * id

Level 1

Level 2

Level 3



$$\begin{aligned}E &\rightarrow E + T \mid T \\T &\rightarrow T * F \mid F \\F &\rightarrow (E) \mid id\end{aligned}$$

Phrase:

- $E + T * id$
- $T * id$
- id

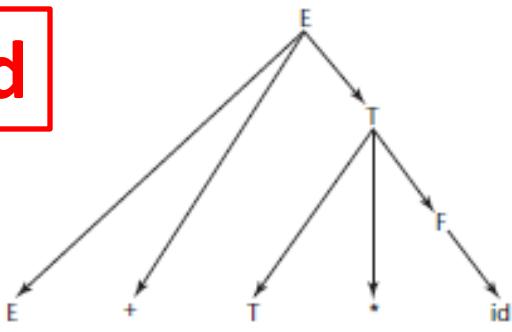
Simple Phrase:

- id

Bottom-Up Parsing - Parsing Problem



id + id * id



$$\begin{aligned}E &\rightarrow E + T \mid T \\T &\rightarrow T * F \mid F \\F &\rightarrow (E) \mid id\end{aligned}$$

Phrase:

- E + T * id
- T * id
- id

Simple Phrase:

- id

The leaves of the parse tree in Figure 4.3 comprise the sentential form $E + T * id$. Because there are three internal nodes, there are three phrases. Each internal node is the root of a subtree, whose leaves are a phrase. The root node of the whole parse tree, E, generates all of the resulting sentential form, $E + T * id$, which is a phrase. The internal node, T, generates the leaves $T * id$, which is another phrase. Finally, the internal node, F, generates id, which is also a phrase. So, the phrases of the sentential form $E + T * id$ are $E + T * id$, $T * id$, and id. Notice that phrases are not necessarily RHSs in the underlying grammar.

The simple phrases are a subset of the phrases. In the previous example, the only simple phrase is id. A simple phrase is always an RHS in the grammar.

The reason for discussing phrases and simple phrases is this: The handle of any rightmost sentential form is its leftmost simple phrase. So now we have a highly intuitive way to find the handle of any right sentential form, assuming we have the grammar and can draw a parse tree. This approach to finding handles is of course not practical for a parser. (If you already have a parse tree, why do you need a parser?) Its only purpose is to provide the reader with some intuitive feel for what a handle is, relative to a parse tree, which is easier than trying to think about handles in terms of sentential forms.

We can now consider bottom-up parsing in terms of parse trees, although the purpose of a parser is to produce a parse tree. Given the parse tree for an entire sentence, you easily can find the handle, which is the first thing to rewrite in the sentence to get the previous sentential form. Then the handle can be pruned from the parse tree and the process repeated. Continuing to the root of the parse tree, the entire rightmost derivation can be constructed.



Bottom-Up Parsing – Parsing Problem

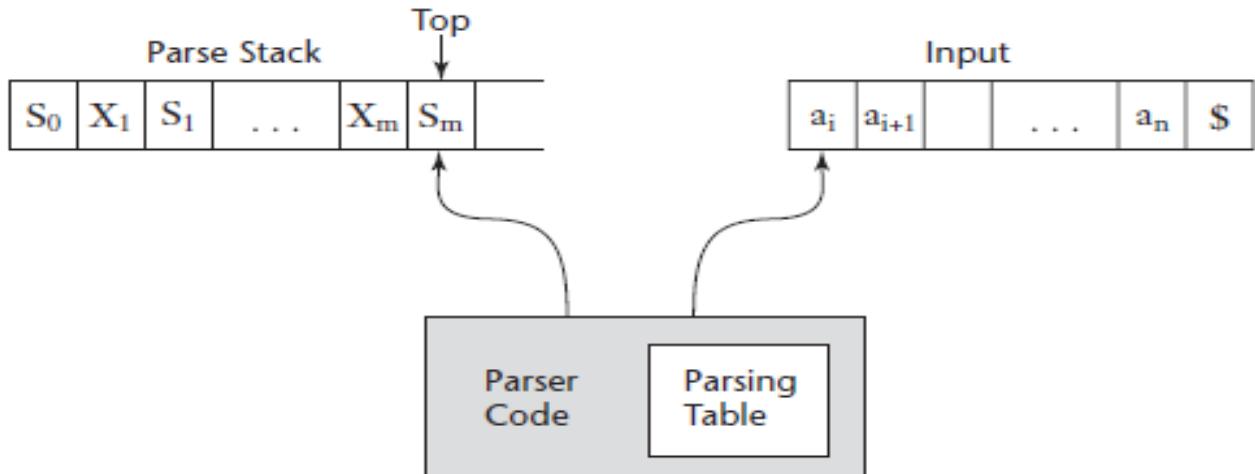
- Phrases are not necessarily RHSs in the underlying grammar
- Simple phrase is always an RHS in the grammar
- Handle of any rightmost sentential form is its leftmost simple phrase
 - Replace id with F
- Consider bottom-up parsing in terms of parse trees
 - Given the parse tree for an entire sentence, you easily can find the handle
 - Rewrite in the sentence and get the previous sentential form
 - Handle can be pruned from the parse tree and the process repeated
- Continue to the root of the parse tree



Shift-Reduce Algorithms

- Bottom-up parsers are often called shift-reduce algorithms
- Two Operations: Shift and Reduce
- Shift -> Moves the next input token onto the parser's stack
- Reduce -> Replaces an RHS (the handle) on top of the parser's stack by its corresponding LHS
- Every parser for a programming language is a pushdown automaton (PDA), because a PDA is a recognizer for a context-free language

PDA



State	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	S5			S4			1	2	3	
1		S6				accept				
2		R2	S7		R2	R2				
3		R4	R4		R4	R4				
4	S5			S4			8	2	3	
5		R6	R6		R6	R6				
6	S5			S4			9	3		
7	S5			S4					10	
8		S6			S11					
9		R1	S7		R1	R1				
10		R3	R3		R3	R3				
11		R5	R5		R5	R5				

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$



Shift-Reduce Algorithms - PDA

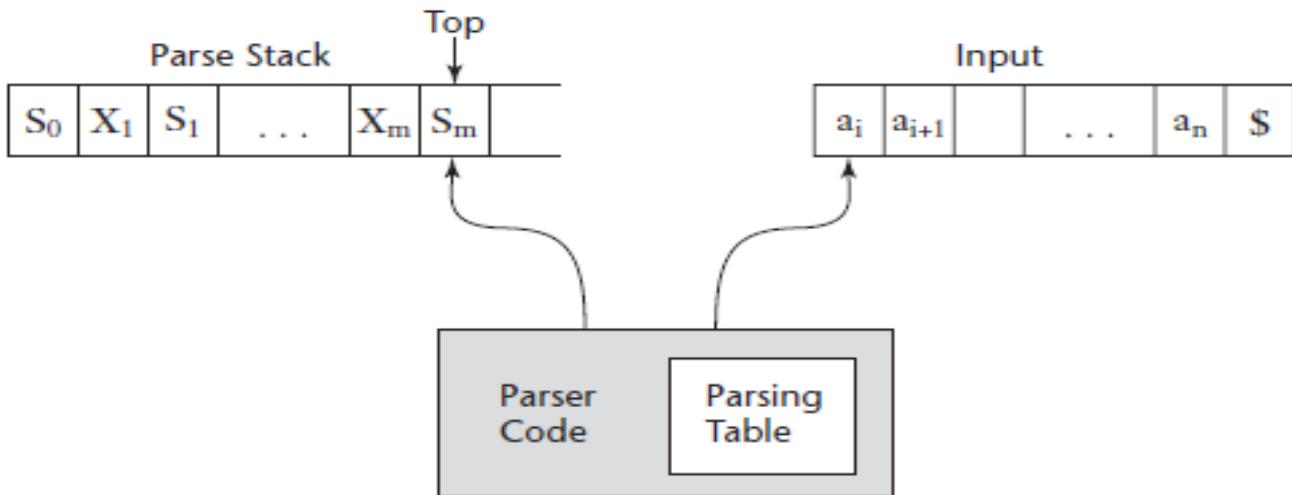
- PDA is a very simple mathematical machine that scans strings of symbols from left to right
- Uses a pushdown stack as its memory
- PDAs can be used as recognizers for context-free languages
- Given a string of symbols over the alphabet of a context-free language, a PDA that is designed for the purpose can determine whether the string is or is not a sentence in the language
- In the process, the PDA can produce the information needed to construct a parse tree for the sentence



LR Parsers

- Many different bottom-up parsing algorithms have been devised
- Most of them are variations of a process called LR
- LR parsers use a relatively small program and a parsing table that is built for a specific programming language
- Advantages of LR Parsers
 - Can be built for all programming languages
 - Can detect syntax errors as soon as it performs a left-to-right scan
 - LR class of grammars is a proper superset of the class parsable by LL parsers
- **Disadv:** Producing the Parsing Table by hand

LR Parsers



- Contents of the parse stack for an LR parser have the following form: $S_0X_1S_1X_2\dots X_mS_m$ (Top)
 - $S \rightarrow$ State Symbols; $X \rightarrow$ Grammar Symbols
 - An LR parser configuration is a pair of strings (Stack, Input):

$$(S_0X_1S_1X_2\dots X_mS_m, a_ia_{i+1}\dots a_n\$)$$



LR Parsers

- Two Parts: ACTION, GOTO
- Action Part
 - State symbols -> Row labels; Terminal symbols -> column labels
- Current parse state + Next input symbol
- Actions: Shift, Reduce, **Accept**, **Error**
 - Shift -> Shifts the next input symbol onto the parse stack
 - Reduce -> Handle on top of the stack, which it reduces to the LHS of the rule whose RHS is the same as the handle

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4			9	3	
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			



LR Parsers

- Goto Part
 - State symbols as labels
 - Has non-terminals as column labels
 - Values in the GOTO part indicate which state symbol should be pushed onto the parse stack after a reduction has been completed
 - The specific symbol is found at the row whose label is the state symbol on top of the parse stack after the handle and its associated state symbols have been removed
 - Column of the GOTO table that is used is the one with the label that is the LHS of the rule used in the reduction

State	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	S5				S4			1	2	3
1		S6					accept			
2		R2	S7			R2	R2			
3		R4	R4			R4	R4			
4	S5				S4			8	2	3
5		R6	R6			R6	R6			
6	S5				S4				9	3
7	S5				S4					10
8		S6				S11				
9		R1	S7			R1	R1			
10		R3	R3			R3	R3			
11		R5	R5			R5	R5			

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

$$\begin{aligned}
 E &\Rightarrow \underline{E} + T \\
 &\Rightarrow E + \underline{T} * F \\
 &\Rightarrow E + \underline{T} * \underline{id} \\
 &\Rightarrow E + \underline{F} * id \\
 &\Rightarrow E + \underline{id} * id \\
 &\Rightarrow \underline{T} + id * id \\
 &\Rightarrow \underline{F} + id * id \\
 &\Rightarrow \underline{id} + id * id
 \end{aligned}$$

Example

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

$$\begin{aligned}
 E &\Rightarrow \underline{E + T} \\
 &\Rightarrow E + \underline{T * F} \\
 &\Rightarrow E + T * \underline{id} \\
 &\Rightarrow E + \underline{F * id} \\
 &\Rightarrow E + \underline{id * id} \\
 &\Rightarrow T + id * id \\
 &\Rightarrow \underline{F + id * id} \\
 &\Rightarrow id + id * id
 \end{aligned}$$

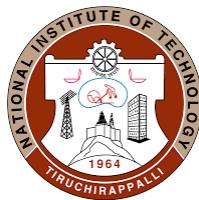
Stack	Input	Action
0	id + id * id \$	Shift 5
0id5	+ id * id \$	Reduce 6 (use GOTO[0, F])
0F3	+ id * id \$	Reduce 4 (use GOTO[0, T])
0T2	+ id * id \$	Reduce 2 (use GOTO[0, E])
0E1	+ id * id \$	Shift 6
0E1+6	id * id \$	Shift 5
0E1+6id5	* id \$	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id \$	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id \$	Shift 7
0E1+6T9*7	id \$	Shift 5
0E1+6T9*7id5	\$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	\$	Reduce 3 (use GOTO[6, T])
0E1+6T9	\$	Reduce 1 (use GOTO[0, E])
0E1	\$	Accept

State	Action						Goto		
	id	+	*	()	\$			
0	S5			S4			1	2	3
1		S6					accept		
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			



Miscellaneous

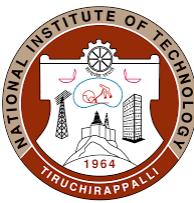
<i>Stack</i>	<i>Input</i>	<i>Action</i>
0	id + id * id \$	Shift 5
0id5	+ id * id \$	Reduce 6 (use GOTO[0, F])
0F3	+ id * id \$	Reduce 4 (use GOTO[0, T])
0T2	+ id * id \$	Reduce 2 (use GOTO[0, E])
0E1	+ id * id \$	Shift 6
0E1+6	id * id \$	Shift 5
0E1+6id5	* id \$	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id \$	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id \$	Shift 7
0E1+6T9*7	id \$	Shift 5
0E1+6T9*7id5	\$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	\$	Reduce 3 (use GOTO[6, T])
0E1+6T9	\$	Reduce 1 (use GOTO[0, E])
0E1	\$	Accept



CSPC31: Principles of Programming Languages

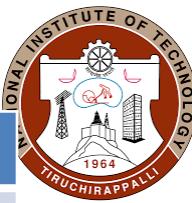
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 6 – Data Types



Objectives

- Introduces the concept of a data type
- Characteristics of common primitive data types
- Arrays, Records and Unions
- Pointers and References
- Design issues and the Design choices made by designers
- Implementation of various data types



Introduction

- Defines a collection of data values and a set of predefined operations
- Language should support an appropriate collection of data types and structures
- Pre-90 Fortrans, Linked Lists and Binary Trees -> Implemented with Arrays
- ALGOL 68 -> Provides a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need
- User-Defined Data Types -> Structure, Union and Enumeration
- Abstract Data Types -> Linked List, Stack and Queue



Introduction

- Structured (non-scalar) Data Types -> Arrays and Records
- How Allocation and Deallocation of memory happens?
- Descriptor -> Collection of the attributes of a variable
 - Descriptor is an area of memory that stores the attributes of a variable
- Attributes are all static -> Descriptors are required only at compile time
 - Built by compiler and are used during compilation
- Dynamic attributes -> Part or all of the descriptor must be maintained during execution
 - Descriptor is used by the run-time system
- Descriptors -> Type checking + Building the code for allocation and deallocation

```
int a;  
float b;
```

Variable Name	Type	Size (in Bits)
a	int	16
b	float	32



Primitive Data Types

- Data types that are not defined in terms of other types are called primitive data types
- Numeric Types
 - Integer
 - Floating Point
 - Complex
 - Decimal
- Boolean Types
- Character Types

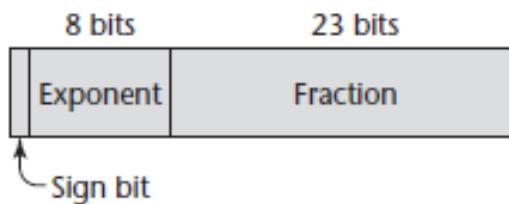


Integer Data Type

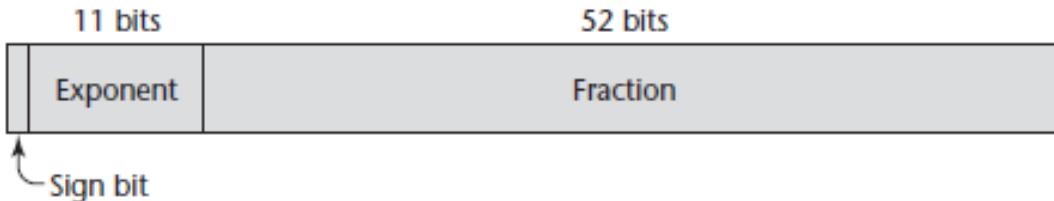
- Byte, Short, Int, Long
- Signed, Unsigned
- Signed
 - MSB = Sign, Other Bits = Value
 - Negative values are stored in 2s Complement form -> Take 1s Complement and then Add 1
- Why not 1s Complement?
 - Negative of an integer is stored as the logical complement of its absolute value. Ones-complement
 - **Disadv:** Has two representations of zero (+0, -0)

Floating-Point Data Type

- Represent real numbers, but the representations are only approximations
 - π, e
- 0.1 in decimal is 0.0001100110011 . . . in binary
- IEEE Floating-Point Standard 754 format.
 - Fractions, Exponent
- Float and Double



← **Float (32 bit)**



← **Double (64 bit)**

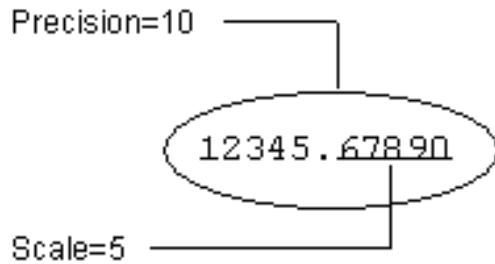


Complex Data Type

- Fortran and Python
- Represented as ordered pairs of floating-point values
- Python -> Imaginary part of a complex literal is specified by using j or J
 - $(7 + 3j)$
- Languages that support a complex type include operations for arithmetic on complex values

Decimal Data Type

- Numeric data type defined by its precision (total number of digits) and scale (number of digits to the right of the decimal point)



- Store a fixed number of decimal digits, with the decimal point at a fixed position in the value
 - COBOL, C#, F#
- 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type
- Computers that are designed to support business systems applications have hardware support for decimal data types

Decimal Data Type

- **Disadv:**

- Range of values is restricted because no exponents are allowed
 - ✓ Precision -> Minimum is 1; maximum is 39
 - ✓ Scale -> Cannot exceed its precision (Can be 0)
- Representation in memory is mildly wasteful
 - ✓ Stored very much like character strings, using binary codes for the decimal digits -> Binary Coded Decimal (BCD)
 - $456 = 010001010110$ (in BCD)
-> Takes more space
 - $456 = 111001000_2$

Decimal	BCD
	8 4 2 1
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Decimal(*p, s*) => Decimal(4, 2) => 10.05

100000011001010101010101010101010.110101001010101010



Boolean Data Type

- Simplest of all types -> True or False
- C89 -> Numeric expressions are used as conditionals
 - All operands with non-zero values -> True; 0 -> False
- Integers can be used, but the use of Boolean is more readable
- Could be represented by a single bit
- Often stored in the smallest efficiently addressable cell of memory, typically a byte



Character Types

- Stored in computers as numeric codings -> 8-bit code ASCII (American Standard Code for Information Interchange)
 - Uses the values 0 to 127 to code 128 different characters
- ISO 8859-1 is another 8-bit character code
 - Allows 256 different characters
- UCS-2 Standard, a 16-bit character set (Unicode)
 - Cyrillic alphabet -> Used in Serbia, Thai digits
- Unicode Consortium + International Standards Organization (ISO) -> 4-byte character code (UCS-4 or UTF-32)

7 (Unused)	6	5	4	3	2	1	0
---------------	---	---	---	---	---	---	---



Character String Types

- Values consist of sequences of characters
- Design Issues:
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?



Strings and their Operations

- Assignment, Catenation, Substring Reference, Comparison and Pattern Matching
- Substring Reference -> Reference to a substring of a given string

`string str = "balakrishnan"`

`string subStr = "krishnan"`

- Substring References are also called as Slices
- Assignment and Comparison -> Complicated with operands of different lengths
- Pattern Matching -> Specified Patterns



Strings and Operations

- C, C++ -> Use Character Arrays (Library -> string.h)
- char str[] = “apples”; -> Stored as “apples0”
- Operations: strcpy, strcmp, strlen (Does not count NULL)
- Parameters and return values for most of the string manipulation functions are char pointers that point to arrays of char
- Problems of string manipulation libraries
 - Strcpy(src, dest); //len(src) = 50; len(dest) = 20
- C++ programmers should use string class from standard library rather than char arrays and C string library



Strings and Operations

- Pattern Matching -> Regular Expressions
- Evolved from UNIX line editor -> ed

`/[A-Za-z][A-Za-z\d]+/ => a1sasdas`

- Brackets enclose character classes
- First character class specifies all letters
- Second specifies all letters and digits (a digit is specified with the abbreviation \d)
- Plus operator following the second category specifies that there must be one or more of what is in the category
- Whole pattern matches strings that begin with a letter, followed by one or more letters or digits



Strings and Operations

`/\d+\.?\d*/` => 234 or 234. or 234.5 or .343

- Pattern matches numeric literals
- `\.` specifies a literal decimal point
- Question mark quantifies what it follows to have zero or one appearance
- Vertical bar (`|`) separates two alternatives in the whole pattern
- First alternative matches strings of one or more digits, possibly followed by a decimal point, followed by zero or more digits
- Second alternative matches strings that begin with a decimal point, followed by one or more digits
- Pattern-matching capabilities using regular expressions are included in the class libraries of C++, Java, Python and C#



Memory Stack

Stack ↓

(Return Address, Argument to Functions, Local Variables, Current CPU State)

Heap ↑

(Dynamic Memory Allocation – malloc, realloc, free)

Global Variables

Program Code



String Length Options

- Static -> Set when the string is created (Static String Length)
- Varying length up to a declared and fixed maximum set by variable's definition -> Limited Dynamic Length Strings
 - Can store any number of characters between zero and the maximum
- Have varying length with no maximum limit (Dynamic Length Strings)
 - overhead of dynamic storage allocation and deallocation, but provides maximum flexibility

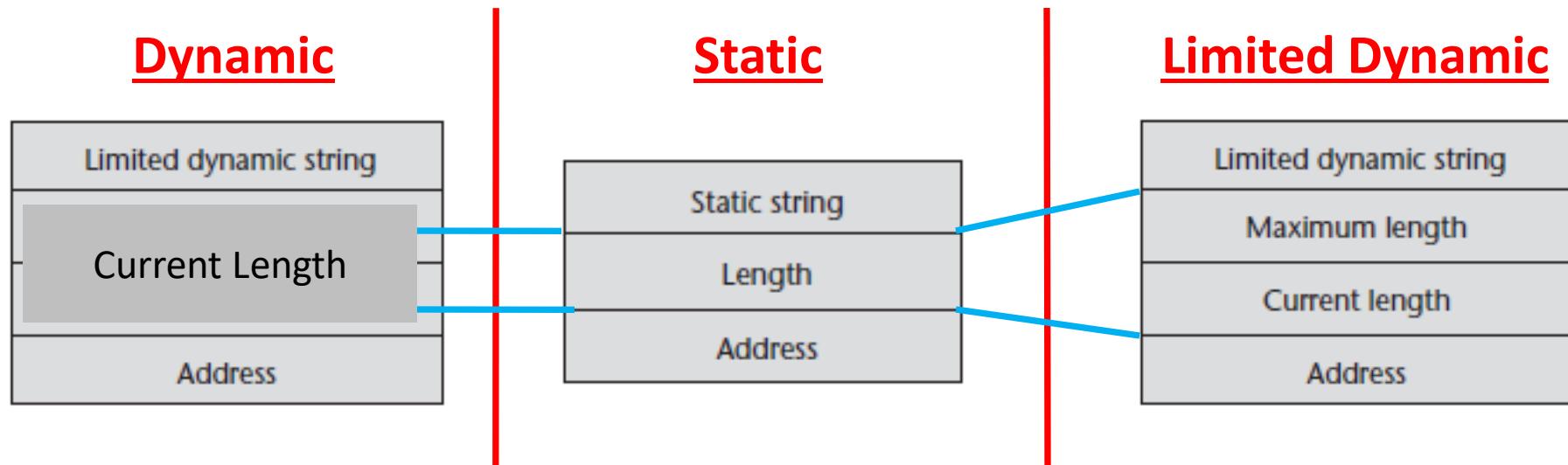


Evaluation

- String types are important to the writability of a language
- Dealing with strings as arrays is not efficient than dealing with a primitive string type
 - strcpy in C -> Simple assignment statement would require a loop
- Providing strings through a standard library is nearly as convenient as having them as a primitive type
- String operations such as simple pattern matching and catenation are essential and should be included for string type values

Implementation of Character String Types

- A descriptor for a static character string type, which is required only during compilation, has three fields



- Limited dynamic strings require a run-time descriptor to store both the fixed maximum length and the current length
- Dynamic length strings require a simpler run-time descriptor because only the current length needs to be stored



Miscellaneous

```
int a;  
float b;
```

Variable Name	Type	Size (in Bits)
a	int	16
b	float	32

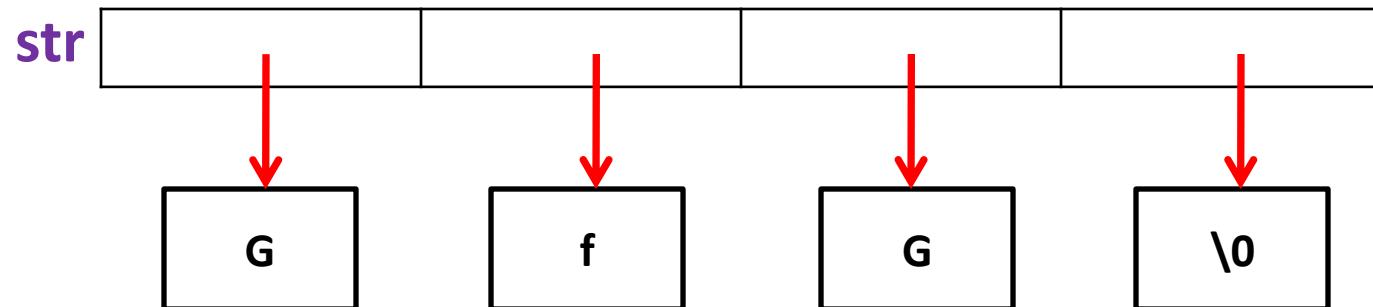


Implementation of Character String Types

- Limited dynamic strings of C and C++
 - Do not require run-time descriptors -> Null Character
 - Do not need the maximum length -> Index values in array references are not range-checked
- Dynamic length strings require more complex storage management -> Grow and Shrink dynamically
- Three approaches
 - Strings can be stored in a linked list -> Heap memory
 - ✓ **Disadv:** Extra memory + Complexity in String Operations
 - Store strings as arrays of pointers to individual characters allocated in Heap
 - ✓ Extra memory + Faster String Operations

Implementation of Character String Types

```
char *str;  
  
int size = 4; /*one extra for '\0'*/  
  
str = (char *)malloc(sizeof(char)*size);  
  
*(str+0) = 'G';  
*(str+1) = 'f';  
*(str+2) = 'G';  
*(str+3) = '\0';
```





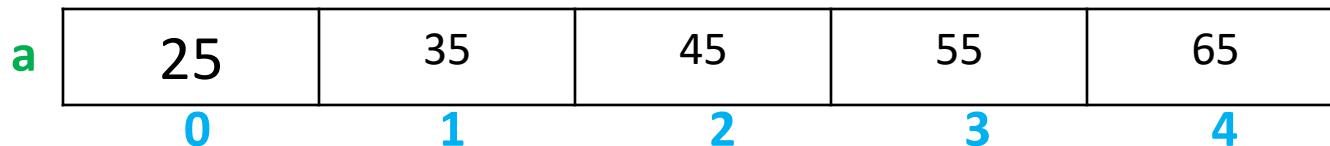
Implementation of Character String Types

- Store complete strings in adjacent storage cells
 - New area of memory is found that can store the complete new string
- Discussion
 - Linked List -> Requires more storage, allocation and deallocation processes are simple
 - ✓ **Disadv:** String operations are slowed by the required pointer chasing
 - Adjacent memory cells -> Faster string operations, Less storage
 - ✓ **Disadv:** Allocation and deallocation processes are slower



Array Types

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



Address of $a[4] = \text{Address of } a[0] + (4 * \text{size of Data Type in Bytes})$

- Individual data elements are of the same type
- References to individual array elements are specified using subscript expressions
- If any of the subscript expressions in a reference include variables, then the reference will require an additional run-time calculation to determine the address of the memory location being referenced

$$x = a[5 + y]$$



Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does array allocation take place?
- Are ragged or rectangular multidimensioned arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- What kinds of slices are allowed, if any?



Arrays and Indices

- Specific elements are referenced by means of a two-level syntactic mechanism -> Name + Subscripts/Indices
 - If all of the subscripts in a reference are constants, the selector is static
- Selection operation can be thought of as a mapping from the array name and the set of subscript values to an element in the aggregate -> Finite Mappings
 - $\text{array_name}(\text{subscript_value_list})$
- In Ada, $\text{Sum} := \text{sum} + \text{B}(I);$ -> Reduces Readability
 - Array element references map the subscripts to a particular element
 - Function calls map the actual parameters to the function definition and, eventually, a functional value



Arrays and Indices

- Two distinct types are involved in an array type: the element type and the type of the subscripts

a[x]

```
type Week_Day_Type is (Monday, Tuesday, Wednesday,  
                         Thursday, Friday);  
type Sales is array (Week_Day_Type) of Float;
```

Sales(Monday) = 6.55; => Sales(0) = 6.55;

- Early programming languages did not specify that subscript ranges must be implicitly checked
 - C, C++, Perl -> Do not
- One can reference an array element in Perl with a negative subscript, Eg: \$list[-2]



Subscript Bindings and Array Categories

- Binding of the subscript type to an array variable is usually static, but the range of values are sometimes dynamically bound
- C -> Lower bound = 0; Fortran 95 -> Lower bound = 1
- Five categories of arrays -> Binding to Subscript ranges, Binding to storage, From where the storage is allocated
 - Static Array -> Subscript ranges are statically bound and storage allocation is static (done before run time)
 - ✓ **Adv:** Efficiency + No dynamic allocation or deallocation is required
 - ✓ **Disadv:** Storage for the array is fixed for the entire execution time of the program



Subscript Bindings and Array Categories

- Fixed stack-dynamic array -> Subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution
 - ✓ **Adv:** Space efficiency -> A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time
 - ✓ **Disadv:** Required allocation and deallocation time
- Stack-dynamic array -> Both subscript ranges and the storage allocation are dynamically bound at elaboration time
 - ✓ Once the values are fixed, they remain unchanged during the lifetime of the variable
 - ✓ **Adv:** Flexibility-> Size need not be known until the array is about to be used



Subscript Bindings and Array Categories

- Fixed-heap dynamic array -> Similar to a fixed stack-dynamic array with few differences
 - ✓ Subscript ranges and the storage binding are both fixed after storage is allocated
 - ✓ Subscript ranges and storage bindings are done when the user program requests them during execution
 - ✓ Storage is allocated from heap rather than from the stack
 - ✓ **Adv:** Array's size always fits the problem
 - ✓ **Disadv:** Allocation time is longer
- Heap-dynamic array -> Binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime
 - ✓ **Adv:** Flexibility -> Grow or shrink
 - ✓ **Disadv:** Allocation and deallocation take longer and may happen many times during execution of the program



Subscript Bindings and Array Categories

- Arrays declared in C and C++ functions that include the static modifier are static
- Arrays that are declared in C and C++ functions (without the static specifier) are examples of fixed stack-dynamic arrays
- C and C++ also provide fixed heap-dynamic arrays
 - Standard C library functions malloc and free, which are general heap allocation and deallocation operations, respectively, can be used for C arrays
- Heterogeneous Arrays -> Elements need not be of the same type
 - Perl, Python, JS, Ruby

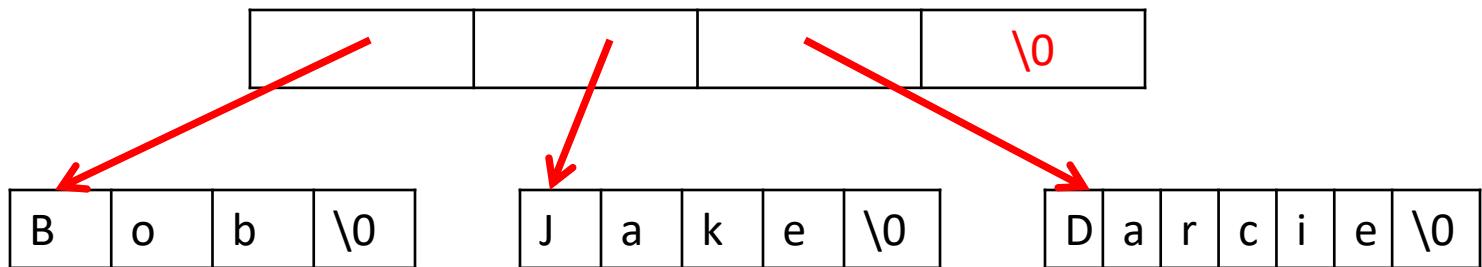
Array Initialization

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

- Compiler sets the length of the array


```
char name [] = "freddie";
```
- Character strings are implemented as arrays of **char**
- Have eight elements, because all strings are terminated with a null character (zero)


```
char *names [] = { "Bob", "Jake", "Darcie"};
```
- Array is one of pointers to characters





Array Operations

- An array operation is one that operates on an array as a unit
- Common operations -> Assignment, Catenation, Comparison for equality and inequality, Slices
- APL
 - Arrays and their operations are the heart of APL
 - It is the most powerful array-processing language ever devised



Array Operations

- In APL, the four basic arithmetic operations are defined for vectors (single-dimensioned arrays) and matrices, as well as scalar operands

$A + B$

- Is a valid expression, whether A and B are scalar variables, vectors, or matrices
- APL includes a collection of unary operators for vectors and matrices (where V is a vector and M is a matrix):

ϕV reverses the elements of V

ϕM reverses the columns of M

θM reverses the rows of M

$\circledcirc M$ transposes M (its rows become its columns and vice versa)

$\div M$ inverts M

Array Operations

- $V = [1 \ 2 \ 3 \ 4 \ 5 \ 6] ; M = [1 \ 2 \ 3$

4 \ 5 \ 6

7 \ 8 \ 9]

- $\phi V = [6 \ 5 \ 4 \ 3 \ 2 \ 1]$

- $\phi M = [7 \ 8 \ 9$

4 \ 5 \ 6

1 \ 2 \ 3]

- $\theta M = [3 \ 2 \ 1$

6 \ 5 \ 4

9 \ 8 \ 7]

- $\odot M = [1 \ 4 \ 7$

2 \ 5 \ 8

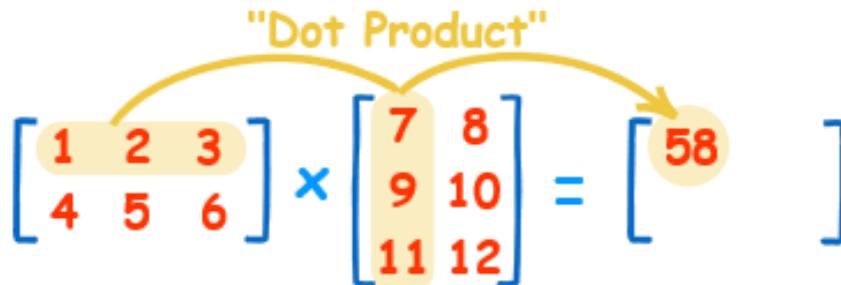
3 \ 6 \ 9]

Array Operations

- If A and B are vectors, $A \times B$ is the mathematical inner product of A and B (a vector of the products of the corresponding elements of A and B)

$$A = [1 \ 2 \ 3]; B = [2 \ 2 \ 2]; A \times B = [2 \ 4 \ 6]$$

- APL includes several special operators that take other operators as Operands
- One of these is the inner product operator, which is specified with a period (.)
 - Takes two operands, which are binary operators, Eg: $+\times$
 - New operator that takes two arguments, either vectors or matrices
 - First multiplies the corresponding elements of two arguments, and then it sums the Results
 - $A +.\times B \rightarrow$ Matrix Multiplication



Rectangular and Jagged Arrays

- Rectangular Array -> Multidimensioned array
 - All of the rows have the same number of elements
 - All of the columns have the same number of elements
 - Rectangular arrays model rectangular tables exactly

a

- C, C++, Java -> $a[2][3]$
- Fortran, Ada and C# -> $a[2, 3]$
- Jagged Array -> Lengths of the rows need not be the same

a



Slices

- Substructure of that array
- It is important to realize that a slice is not a new data type
- Mechanism for referencing part of an array as a unit
- If arrays cannot be manipulated as units in a language, that language has no use for slices

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- vector[3:6] -> Three-element array with the fourth through sixth elements of vector
- mat[1] -> Second row of mat
- mat[0][0:2] -> First and Second element of the first row of mat, which is [1, 2]



Implementation of Array Types

- Implementing arrays requires considerably more compile-time effort than does implementing primitive types
- Code to allow accessing of array elements must be generated at compile time
- At run time, this code must be executed to produce element addresses
- No way to precompute the address to be accessed by a reference, $\text{list}[k]$

$$\text{address}(\text{list}[k]) = \text{address}(\text{list}[0]) + k * \text{element_size}$$

- If the element type is statically bound and the array is statically bound to storage, then the value of the constant part can be computed before run time

$$\text{address}(\text{list}[k]) = \text{address}(\text{list}[\text{lower_bound}]) + ((k - \text{lower_bound}) * \text{element_size})$$



Implementation of Array Types

- Compile-time descriptor for single-dimensioned arrays includes information required to construct the access function
- If run-time checking of index ranges is not done and the attributes are all static, then only the access function is required during execution; no descriptor is needed
- If run-time checking of index ranges is done, then those index ranges may need to be stored in a run-time descriptor
- If the subscript ranges of a particular array type are static, then the ranges may be incorporated into the code that does the checking, thus eliminating the need for the run-time descriptor
- If any of the descriptor entries are dynamically bound, then those parts of the descriptor must be maintained at run time

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



Implementation of Array Types

- Two ways in which multidimensional arrays can be mapped to one dimension: row major order and column major order

3 4 7

6 2 5

1 3 8

- Row Major Order -> 3, 4, 7, 6, 2, 5, 1, 3, 8
- Column Major Order (Fortran) -> 3, 6, 1, 4, 2, 3, 7, 5, 8

Implementation of Array Types



0	1	...	$j-1$	j	...	$n-1$
1						
:						
$i-1$						
i				\otimes		
:						
$m-1$						

$$\begin{aligned} & (((i - \text{row_lb}) * n) + (j - \text{col_lb})) * \text{element_size} \\ &= (\mathbf{i * n * element_size}) - (\text{row_lb} * n * \text{element_size}) + \\ & \quad (\mathbf{j * element_size}) - (\text{col_lb} * \text{element_size}) \end{aligned}$$

$\text{location}(a[i, j]) = \text{address of } a[0, 0]$
+ (((number of rows above the i^{th} row) * (size of a row))
+ (number of elements left of the j^{th} column)) *
element size)

$\text{location}(a[i, j]) = \underline{\text{address of } a[0, 0]} + \underline{(((i * n) + j) *}$
element_size)

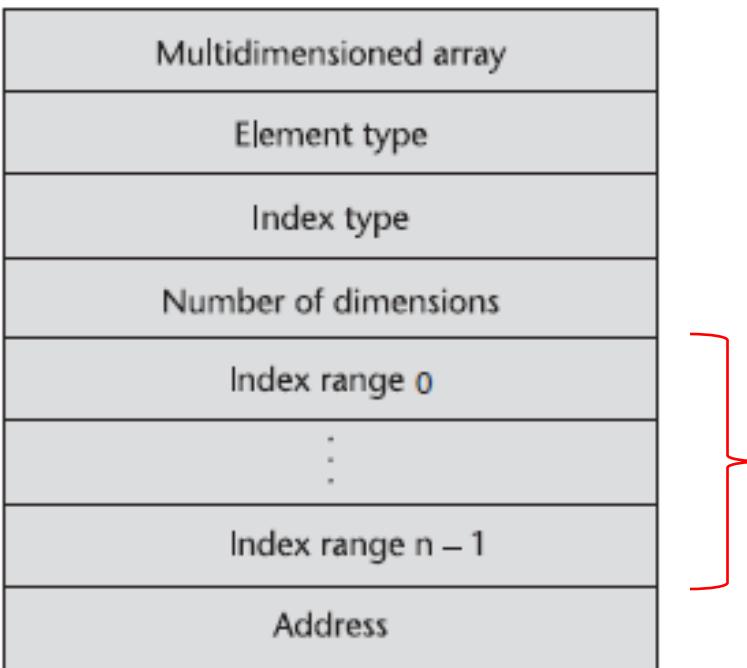
$\text{location}(a[i, j]) = \text{address of } a[\text{row_lb}, \text{col_lb}]$
+ ((($i - \text{row_lb}$) * n) + ($j - \text{col_lb}$)) * element_size

$\text{location}(a[i, j]) = \underline{\text{address of } a[\text{row_lb}, \text{col_lb}]}$
- (((row_lb * n) + col_lb) * element_size)
+ ((($i * n$) + j) * element_size)

Implementation of Array Types



- For each dimension of an array, one add and one multiply instruction are required for the access function
- Accesses to elements of arrays with several subscripts are costly

$$\text{address}(\text{list}[k]) = \text{address}(\text{list}[\text{lower_bound}]) + ((k - \text{lower_bound}) * \text{element_size})$$
$$\begin{aligned}\text{location}(a[i, j]) &= \text{address of } a[\text{row_lb}, \text{col_lb}] \\ &\quad + (((i - \text{row_lb}) * n) + (j - \text{col_lb})) * \text{element_size}\end{aligned}$$




Associative Arrays

- Unordered collection of data elements that are indexed by an equal number of values called keys

Key	Value
0	100
2	20
4	30
1	30
3	10

- Non-associative arrays, the indices never need to be stored (because of their regularity)

a	100	30	20	10	30
	0	1	2	3	4



Structure and Operations

Perl:

```
%salaries = ("Gary" => 75000, "Perry" => 57000,  
             "Mary" => 55750, "Cedric" => 47850);  
  
$salaries{"Perry"} = 58850;  
  
delete $salaries{ "Gary" } ;  
  
@salaries = ();  
  
if (exists $salaries{ "Shelly" }) . . .
```



Structure and Operations

- *keys* operator -> Returns an array of the keys
- *values* operator -> Returns an array of the values
- *each* operator -> Iterates over the element pairs
- Hash is much efficient
 - If searches of the elements are required
 - Data to be stored is paired

Name	Salary
Bala	10,000/-
Krishnan	20,000/-

- If every element of the list is to be processed, it is more efficient to use an array



Implementing Associative Arrays

- Perl
 - 32-bit hash value is computed for each entry and is stored with the entry
 - When an associative array must be expanded beyond its initial size, the hash function need not be changed; rather, more bits of the hash value are used
 - Only half of the entries must be moved when this happens
 - So, although expansion of an associative array is not free, it is not as costly as might be expected
- PHP
 - Elements in arrays are placed in memory through a hash function
 - All elements are linked together in the order in which they were created
 - Links are used to support iterative access to elements through the current and next functions



Record Types

- Aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure
 - Information about a college student -> Name, #, GPA

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```



Record Types

- Design Issues
 - What is the syntactic form of references to fields
 - Are elliptical references allowed?

Definitions of Records



Sl. No.	Records	Arrays
1.	Heterogenous	Homogenous
2.	References are made using identifiers	References are made using indices
3.	Allowed to include Unions	Not possible

EMPLOYEE-RECORD record consists of the EMPLOYEE-NAME record and the HOURLY-RATE field. The numerals 01, 02, and 05 that begin the lines of the record declaration are level numbers, which indicate by their relative values the hierarchical structure of the record. Any line that is followed by a line with a higher-level number is itself a record.

COBOL

```

01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST      PICTURE IS X(20).
        05  MIDDLE     PICTURE IS X(10).
        05  LAST       PICTURE IS X(20).
    02  HOURLY-RATE PICTURE IS 99V99.

```

Ada

```

type Employee_Name_Type is record
    First : String (1..20);
    Middle : String (1..10);
    Last : String (1..20);
end record;
type Employee_Record_Type is record
    Employee_Name: Employee_Name_Type;
    Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;

```



References to Record Fields

COBOL

field_name OF record_name_1 OF . . . OF record_name_n

MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD

Other

Employee_Record.Employee_Name.Middle

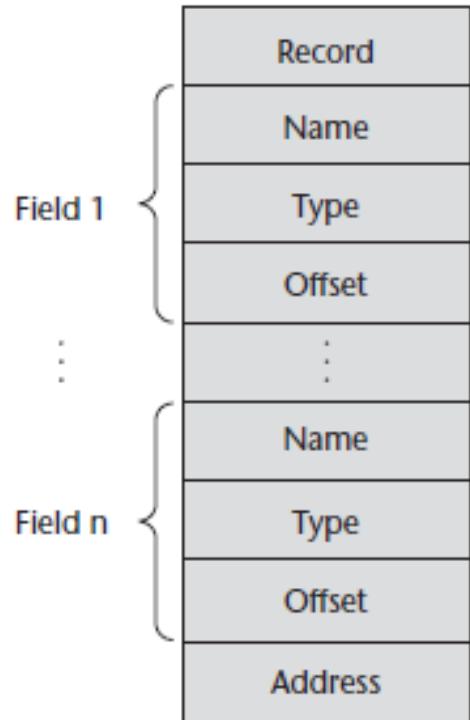
- Fully Qualified Reference -> All intermediate record names from the largest enclosing record to the specific field are named in the reference
- Elliptical Reference -> Any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous
 - FIRST, FIRST OF EMPLOYEE-NAME, and FIRST OF EMPLOYEE-RECORD are elliptical references

```
01 EMPLOYEE-RECORD.  
  02 EMPLOYEE-NAME.  
    05 FIRST      PICTURE IS X(20).  
    05 MIDDLE     PICTURE IS X(10).  
    05 LAST       PICTURE IS X(20).  
  02 HOURLY-RATE PICTURE IS 99V99.
```

Implementation of Record Types

- Fields of records are stored in adjacent memory locations
- Access method used for arrays cannot be used for records
 - Offset address, relative to the beginning of the record, is associated with each field

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```



- Field accesses are all handled using these offsets

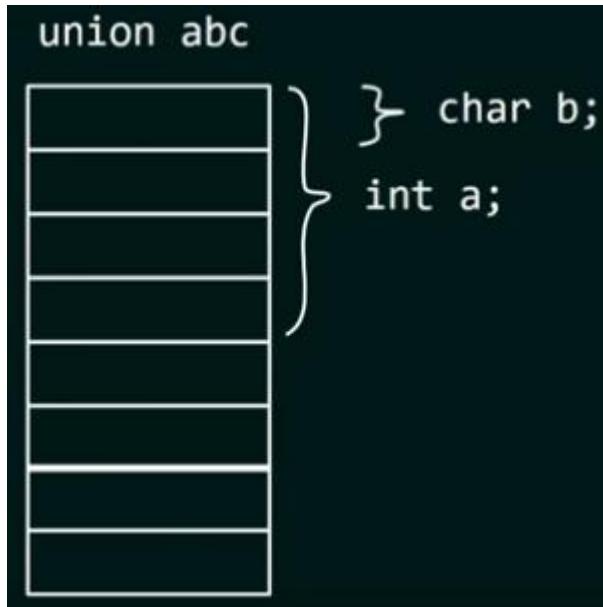
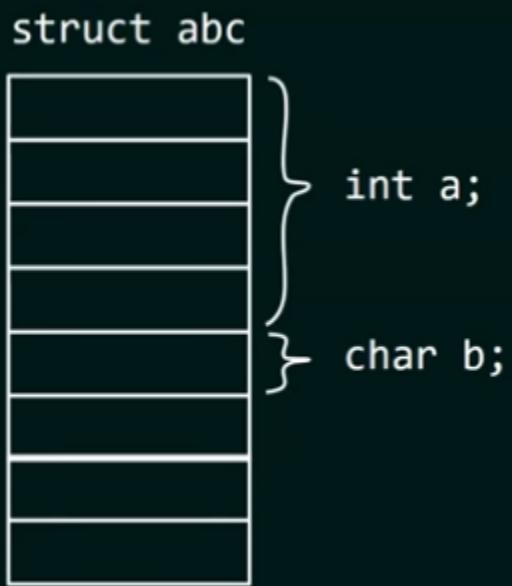
Struct vs Union

```
struct abc {  
    int a;  
    char b;  
};
```

a's address = 6295624
b's address = 6295628

```
union abc {  
    int a;  
    char b;  
};
```

a's address = 6295616
b's address = 6295616





Union Types

- A type whose variables may store different type values at different times during program execution

union Data

```
{  
    int i;           //2 Bytes  
    float f;        //4 Bytes  
    char str[20];   //20 Bytes  
} data;
```

- Share memory
- Required size is 20 Bytes

- Design issues

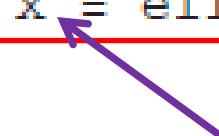
- Should type checking be required?
- Should unions be embedded in records?



Discriminated vs Free Unions

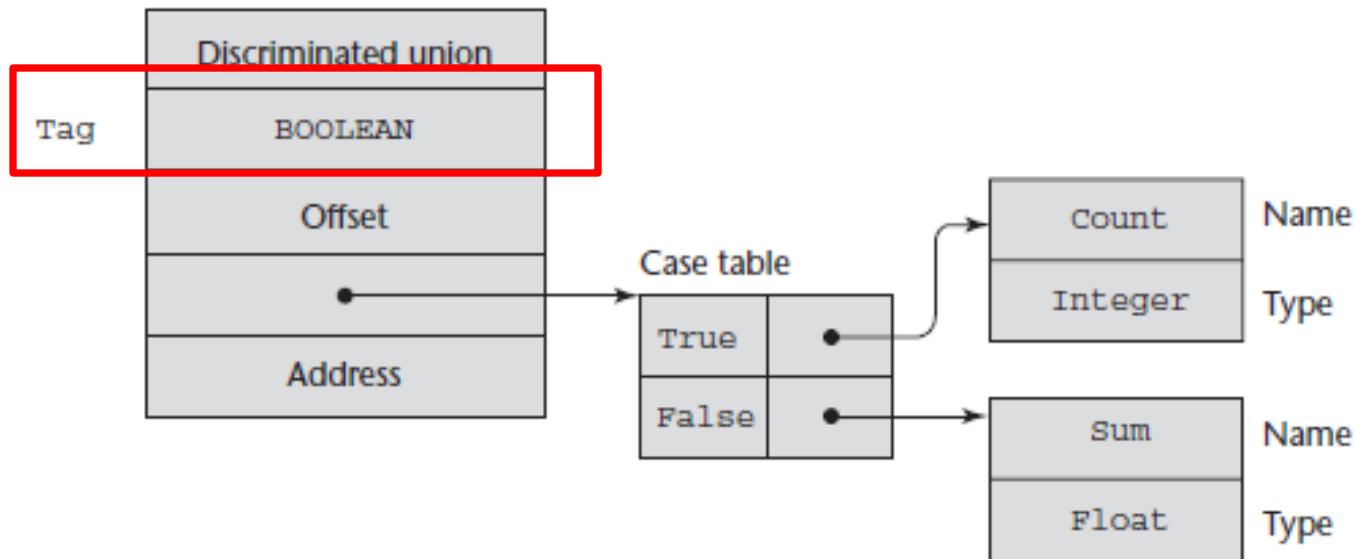
- Fortran, C, C++ -> No type checking
 - Free Unions
- Last assignment is not type checked
 - System cannot determine the current type of the current value of ell
 - Assigns the bit string representation of 27 to the float variable x

```
union flexType {  
    int intEl;  
    float floatEl;  
};  
  
union flexType ell;  
float x;  
...  
  
ell.intEl = 27;  
x = ell.floatEl;
```



Discriminated vs Free Union

- Type checking of unions requires that each union construct include a type indicator
- Such an indicator is called a tag or discriminant
- Union with a discriminant is called a discriminated union
- First language to support is ALGOL68 -> Later Ada





Ada Union Types

- Constrained Variant Variable -> Allows to specify variables of a variant record type that will store only one of the possible type values in the variant
 - Type checking can be static
- Unconstrained Variant Variable -> Allow the values of their variants to change types during execution
 - Type of the variant can be changed only by assigning the entire record, including the discriminant

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 =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
      when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
  end record;
  
```

```

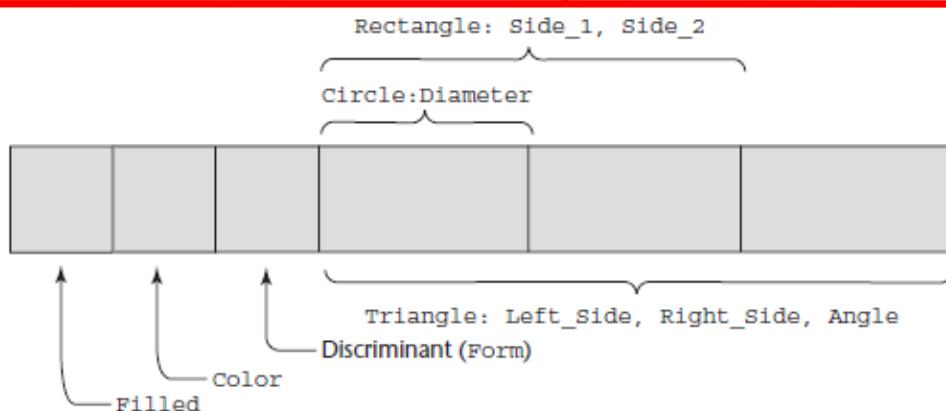
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
  
```

Figure_1 is declared to be an unconstrained variant record that has no initial value. Its type can change by assignment of a whole record, including the discriminant

```

Figure_1 := (Filled => True,
              Color => Blue,
              Form => Rectangle,
              Side_1 => 12,
              Side_2 => 3);

if (Figure_1.Diameter > 3.0) ...
  
```





Evaluation

- Unions are potentially unsafe constructs in some languages like C and C++
 - Not strongly typed -> These languages do not allow type checking of references to their unions
- Unions can be safely used in Ada

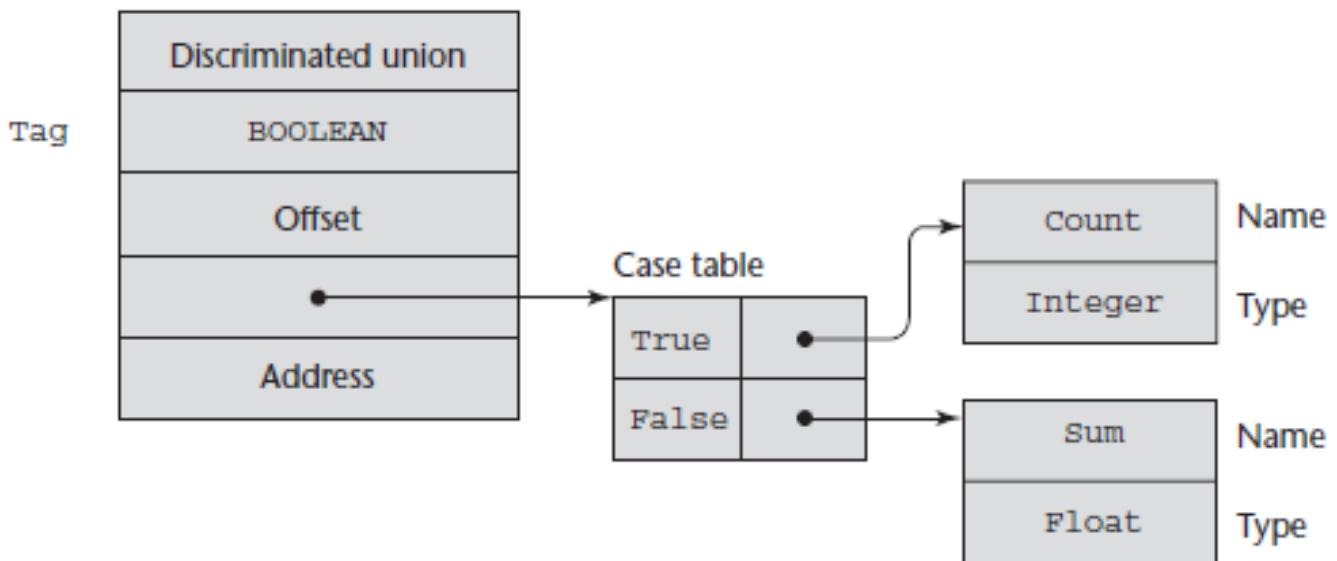


Implementation

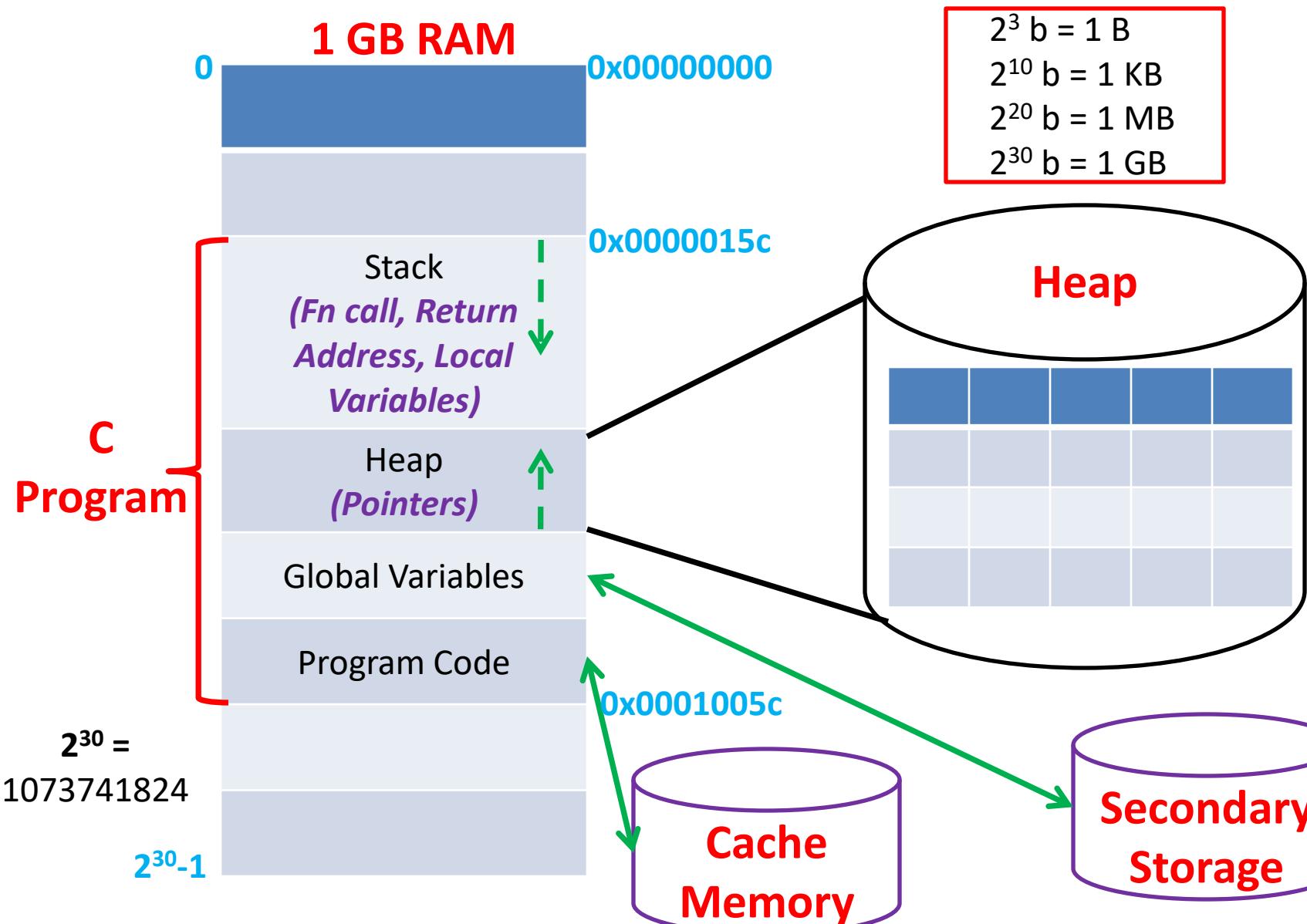
- Unions are implemented by simply using the same address for every possible variant
- Sufficient storage for the largest variant is allocated
- Tag of a discriminated union is stored with the variant in a record-like structure
- At compile time, the complete description of each variant must be stored
 - Can be done by associating a case table with the tag entry in the descriptor
 - Case table has an entry for each variant, which points to a descriptor for that particular variant

Implementation

```
type Node (Tag : Boolean) is
  record
    case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
    end case;
  end record;
```



Memory Management





Binary to Hex

00 0000 1010 0010₂

0000 0000 1010 0010₂

0 0 A 2₁₆

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F



Pointer vs Reference

- Pointer Variable

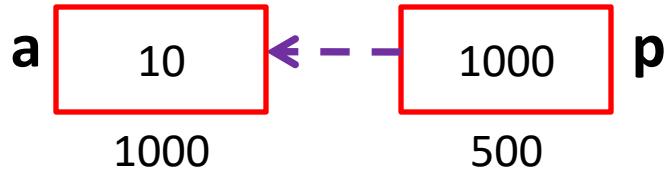
```
int a = 10;
```

```
int *p = &a;
```

```
int a = 10;
```

```
int *p;
```

```
p = &a;
```

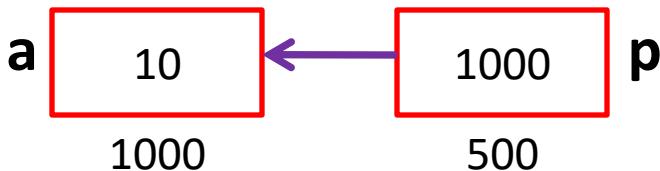


Initialization
Reassignment
NULL Value
Indirection
Arithmetic Operations

- Reference Variable

```
int a=10;
```

```
int & p=a;
```





Pointer vs Reference

```
#include <iostream>
using namespace std;
void square(int &);
int main()
{
    int number = 8;
    cout << "In main(): " << &number << endl;      // 0x22ff1c
    cout << number << endl;                      // 8
    square(number);
    cout << number << endl;                      // 8
}

void square(int rNumber)                         // New Local variable rNumber will be created
{
    cout << "In square(): " << &rNumber << endl; // 0x22ff1f
    rNumber *= rNumber;
}
```

Pass-by-Value



Pointer vs Reference

```
#include <iostream>
using namespace std;
void square(int *);
int main()
{
    int number = 8;
    cout << "In main(): " << &number << endl;      // 0x22ff1c
    cout << number << endl;                      // 8
    square(&number);                                // Explicit referencing to pass an address
    cout << number << endl;                      // 64
}
void square(int * pNumber)                         // Function takes an int pointer
{
    cout << "In square(): " << pNumber << endl;    // 0x22ff1c
    *pNumber *= *pNumber;                            // Explicit de-referencing to get the value pointed-to
}
```

Pass-by-Reference with Pointer Arguments



Pointer vs Reference

```
#include <iostream>
using namespace std;
void square(int &); // Function takes an int reference
int main()
{
    int number = 8;
    cout << "In main(): " << &number << endl; // 0x22ff1c
    cout << number << endl; // 8
    square(number); // Implicit referencing (without '&')
    cout << number << endl; // 64
}

void square(int & rNumber) // Function takes an int reference
{
    cout << "In square(): " << &rNumber << endl; // 0x22ff1c
    rNumber *= rNumber; // Implicit de-referencing (without '*')
}
```

Pass-by-Reference with Reference Arguments



Static vs Dynamic Memory Allocation

// Static Allocation

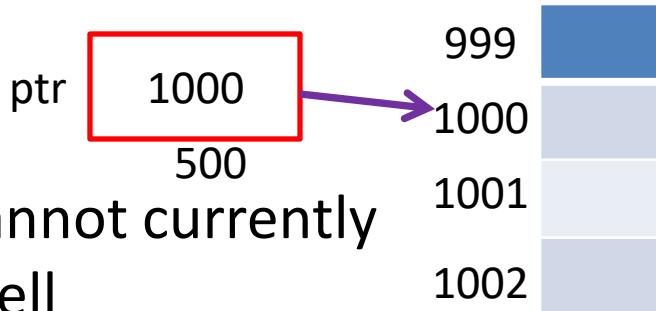
```
int number = 88;  
int * p1 = &number;           // Assign a "valid" address into pointer
```

// Dynamic Allocation

```
int * p2;                 // Not initialize, points to somewhere which is invalid  
cout << p2 << endl;       // Print address before allocation  
p2 = new int;             // Dynamically allocate an int and assign its address to pointer  
                          // The pointer gets a valid address with memory allocated  
*p2 = 99;  
cout << p2 << endl;       // Print address after allocation  
cout << *p2 << endl;       // Print value point-to  
delete p2;                // Remove the dynamically allocated storage
```

Pointer and Reference Types

- A pointer type is one in which the variables have a range of values that consists of memory addresses and a special value, NIL (NULL/0)
- NIL is not a valid address
 - Used to indicate that a pointer cannot currently be used to reference a memory cell
- Purpose
 - Indirect addressing
 - Provide a way to manage dynamic storage. It can be used to access a location in an area where storage is dynamically allocated called a heap





Pointer and Reference Types

- Variables that are dynamically allocated from the heap are called heap-dynamic variables
- They often do not have identifiers associated with them
 - Can be referenced only by pointer or reference type variable
- Variables without names -> Anonymous Variables
- Pointers, unlike arrays and records, are not structured types
- Also different from scalar variables -> Used to reference some other variable (Reference Type) rather than being used to store data (Value Type)
- Use of pointers add writability to a language



Design Issues

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable (the value a pointer references)?
- 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

- Operations: Assignment and Dereferencing

```
int var = 20;
```

```
int *ip;
```

```
ip = &var;
```

```
int x = *ip;
```

- Assignment Operation -> Sets a pointer variable's value to some useful address

- If pointer variables are used only to manage dynamic storage, then the allocation mechanism, whether by operator or built-in subprogram, serves to initialize the pointer variable
- If pointers are used for indirect addressing to variables that are not heap dynamic, then there must be an **explicit operator** or built-in subprogram for fetching the address of a variable, which can then be assigned to the pointer variable

Pointer Operations

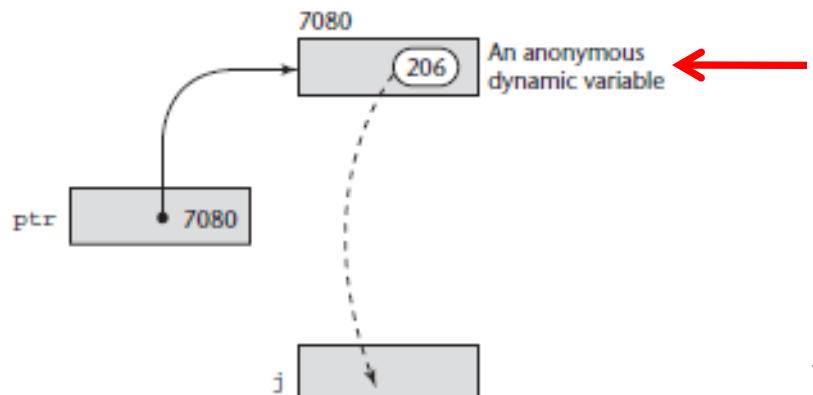
- An occurrence of a pointer variable in an expression can be interpreted in two distinct ways
 - Reference to the contents of the memory cell to which it is bound, which in the case of a pointer is an address -> Normal Reference Pointer

```
int var = 20;
int *ip;
ip = &var;
```



- Reference to the value in the memory cell pointed to by the memory cell to which the pointer variable is bound. In this case, the pointer is interpreted as an indirect reference -> Dereferencing the Pointer

`j = *ptr // Sets j = 206`





Pointer Operations

- When pointers point to records, the syntax varies among languages
- C, C++ -> Two ways a pointer to a record can be used to reference a field in that record
 - $p \rightarrow \text{age}; (*p).\text{age}$
- Languages that provide pointers for the management of a heap must include an explicit allocation and deallocation operation
 - `malloc, new`
 - `delete`



Pointer Problems

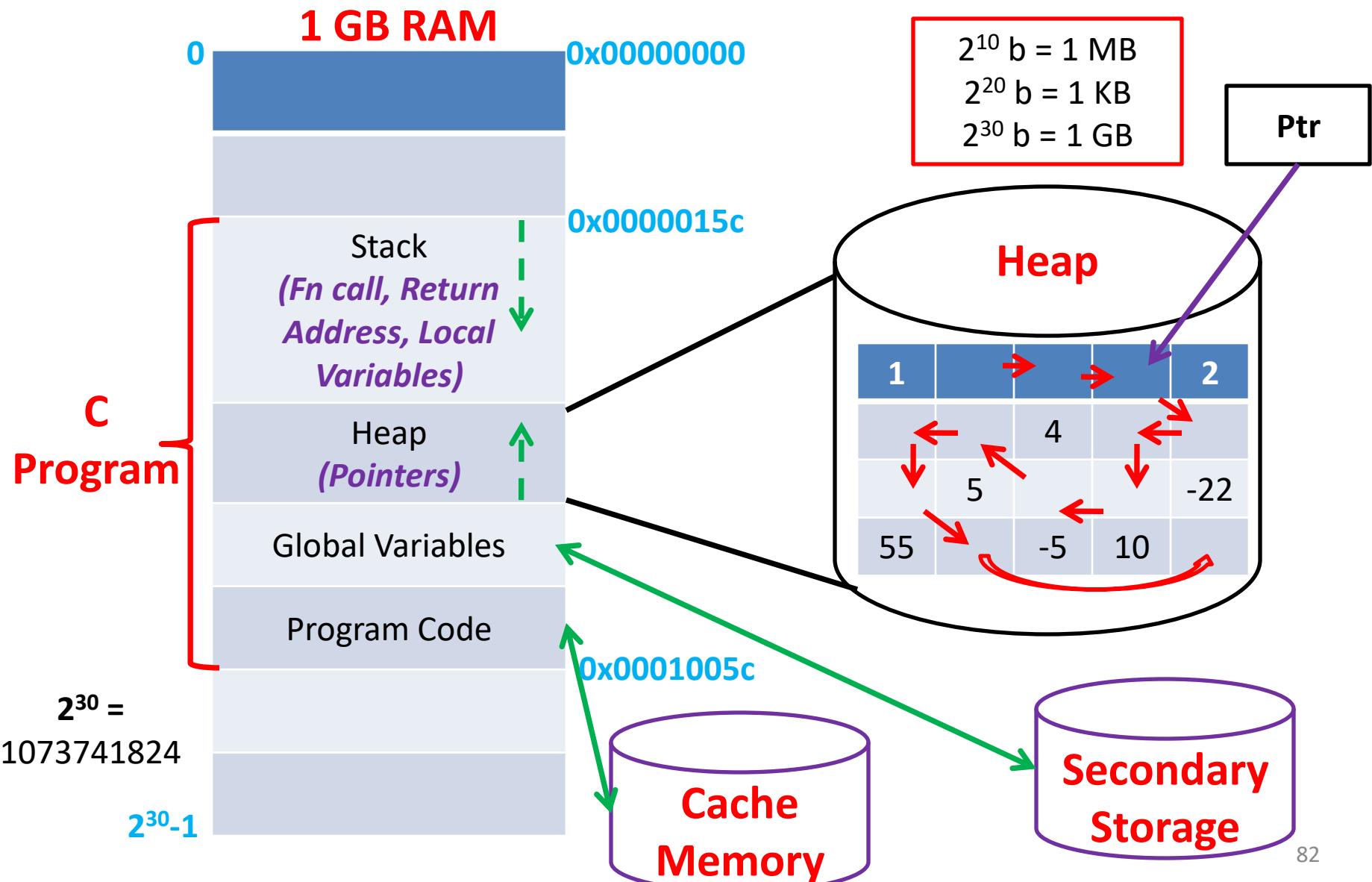
- First high-level programming language to include pointer variables was PL/I
 - Pointers could be used to refer to both heap-dynamic variables and other program variables
 - Were highly flexible, but their use could lead to several kinds of programming errors
- Some of the problems of PL/I pointers are also present in the pointers of subsequent languages
- Recent languages, such as Java, have replaced pointers completely with reference types
- A reference type is really only a pointer with restricted operations



Dangling Pointers

- Also called as dangling reference
- Pointer that contains the address of a heap-dynamic variable that has been deallocated
- Problems
 - Location being pointed to may have been reallocated to some new heap-dynamic variable
 - If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid
 - Even if the new dynamic variable is the same type, its new value will have no relationship to the old pointer's dereferenced value
 - Furthermore, if the dangling pointer is used to change the heap-dynamic variable, the value of the new heap-dynamic variable will be destroyed
 - Finally, it is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail

Memory Management





Dangling Pointers

- Possible ways to create Dangling Pointers
 - A new heap-dynamic variable is created and pointer p2 is set to point at it
 - Pointer p1 is assigned p2's value
 - The heap-dynamic variable pointed to by p2 is explicitly deallocated (possibly setting p2 to nil), but p1 is not changed by the operation
 - p1 is now a dangling pointer
 - If the deallocation operation did not change p2, both p1 and p2 would be dangling. (Of course, this is a problem of aliasing—p1 and p2 are aliases)

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Now, arrayPtr1 is dangling, because the heap storage
// to which it was pointing has been deallocated.
```



Lost Heap-Dynamic Variables

- An allocated heap-dynamic variable that is no longer accessible to the user program
- Such variables are often called garbage, because they are not useful for their original purpose, and they also cannot be reallocated for some new use in the program
- Possible way to create
 - Pointer p1 is set to point to a newly created heap-dynamic variable
 - p1 is later set to point to another newly created heap-dynamic variable
- The first heap-dynamic variable is now inaccessible or lost -> Memory Leakage
- Memory leakage is a problem, regardless of whether the language uses implicit or explicit deallocation



Pointers in C and C++

- Can point anywhere in memory, whether there is a variable there or not, which is one of the dangers of such pointers
- Asterisk (*) denotes the dereferencing operation
- Ampersand (&) denotes the operator for producing the address of a variable

```
int *ptr;  
int count, init;  
...  
ptr = &init;  
count = *ptr;
```

count = init;

- Pointers can be assigned the address value of any variable of the correct domain type, or they can be assigned the constant zero, which is used for nil.



Pointers in C and C++

ptr + index

- Instead of simply adding the value of index to ptr, the value of index is first scaled by the size of the memory cell (in memory units) to which ptr is pointing (its base type)

```
int list [10];
```

```
int *ptr;
```

```
ptr = list; // Assigns the address of list[0] to ptr
```

- $*(\text{ptr} + 1)$ is equivalent to $\text{list}[1]$
- $*(\text{ptr} + \text{index})$ is equivalent to $\text{list}[\text{index}]$
- $\text{ptr}[\text{index}]$ is equivalent to $\text{list}[\text{index}]$
- Pointers in C and C++ can point to functions
 - Used to pass functions as parameters to other functions



Pointers in C and C++

- C and C++ include pointers of type void *, which can point at values of any type -> Generic Pointers
 - Type checking is not a problem with void * pointers, because these languages disallow dereferencing them

```
int a = 10;      void *ptr = &a;      printf("%d", *ptr);  
// printf("%d", *(int *)ptr);
```

```
int a = 10;  
char b = 'x';  
void *p = &a;  
p = &b;
```

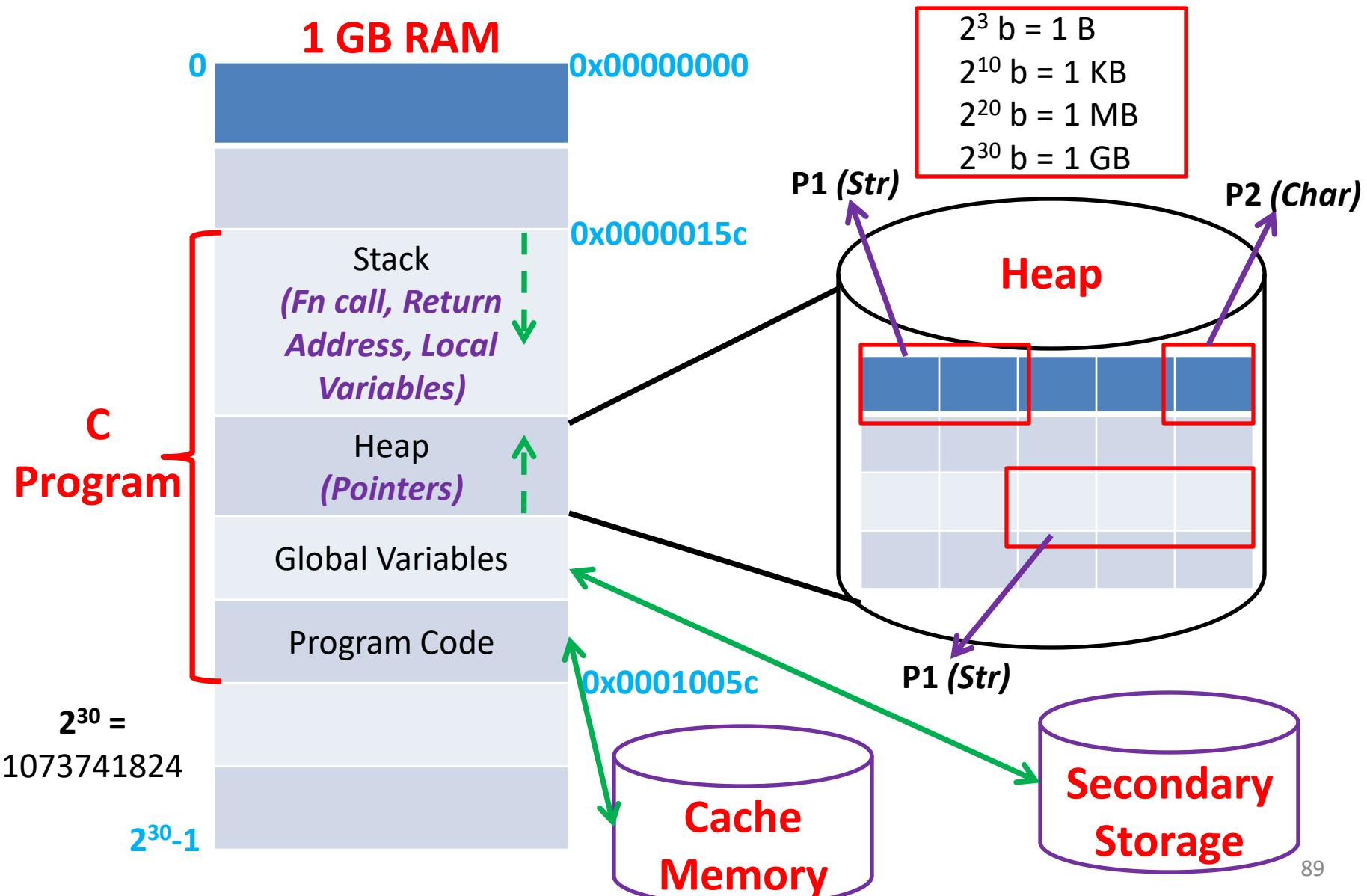
```
int main(void)  
{  
    // Note that malloc() returns void * which can be  
    // typecasted to any type like int *, char *, ..  
    int *x = malloc(sizeof(int) * n);  
}
```



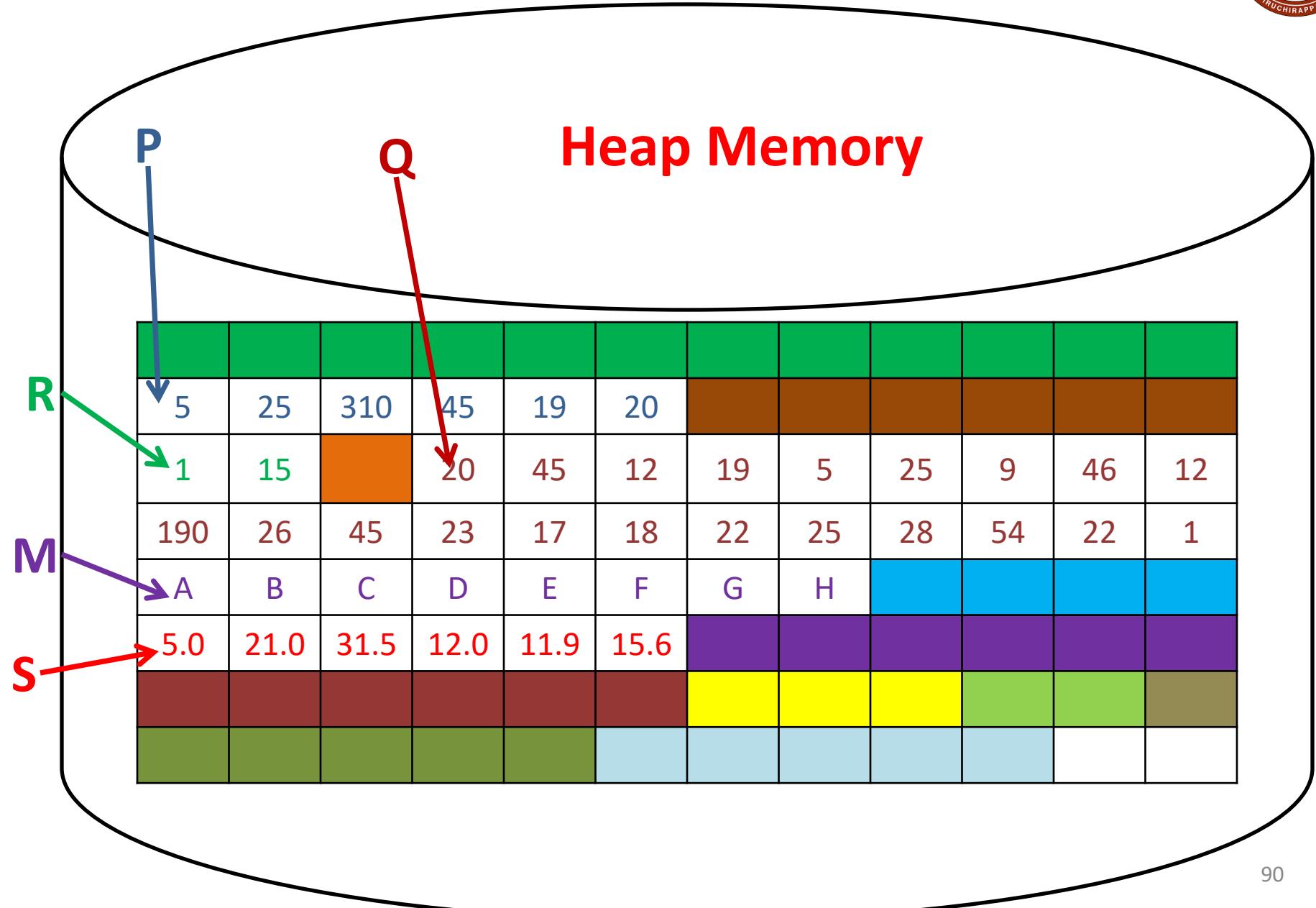
Pointers in C and C++

- Common use of void * pointers is as the types of parameters of functions that operate on memory
- **Eg:** Suppose we wanted a function to move a sequence of bytes of data from one place in memory to another
- It would be most general if it could be passed two pointers of any type
- This would be legal if the corresponding formal parameters in the function were void * type
- The function could then convert them to char * type and do the operation, regardless of what type pointers were sent as actual parameters

Memory Management



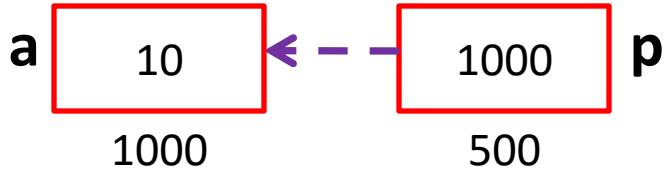
Miscellaneous



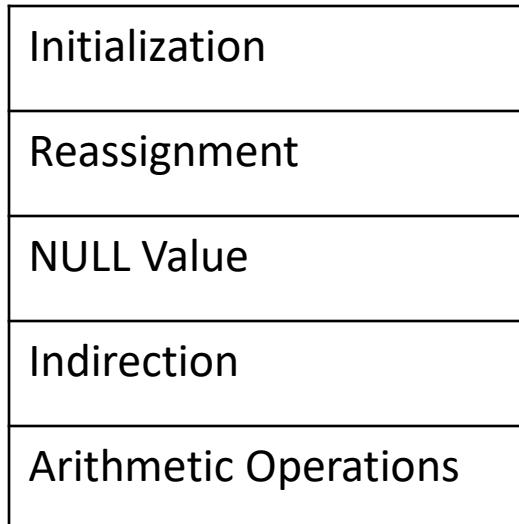
Pointer vs Reference

- Pointer Variable

```
int a = 10;
int *p = &a;
```

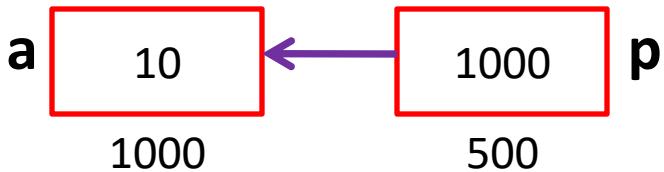


```
int a = 10;
int *p;
p = &a;
int j = p;
int j = *p;
```



- Reference Variable

```
int a=10;
int & p=a;
int j = p;
```





Reference Types

- A pointer refers to an address in memory, while a reference refers to an object or a value in memory
- Although it is natural to perform arithmetic on addresses, it is not sensible to do arithmetic on references
`ptr + index // Valid for Pointers and not for References`
- C++ includes a special kind of reference type that is:
 - Used primarily for the formal parameters in function definitions
 - A constant pointer and is always implicitly dereferenced
 - Must be initialized with the address of some variable in its definition
 - After initialization a reference type variable can never be set to reference any other variable
- The implicit dereference prevents assignment to the address value of a reference variable

```
int result = 0;  
int &ref_result = result;  
...  
ref_result = 100;
```

result and ref_result
are aliases



Evaluation

- **Problems:** Dangling Pointers; Garbage
- Pointers have been compared with the goto
 - Goto statement widens the range of statements that can be executed next
 - Pointer variables widen the range of memory cells that can be referenced by a variable
- **Application:** Pointers are necessary to write device drivers, in which specific absolute addresses must be accessed



Implementation of Pointer and Reference Types

- Pointers are used in heap management
 - First, describe how pointers and references are represented internally
- Discuss two possible solutions to the dangling pointer problem
- Finally, describe the major problems with heap-management techniques



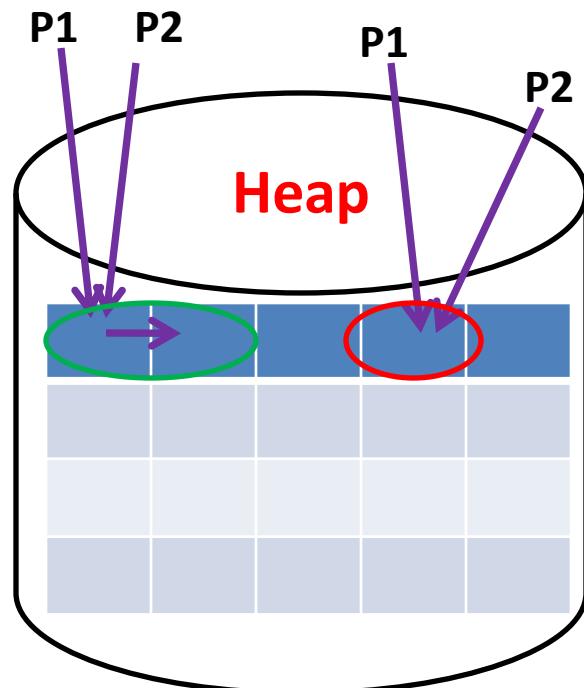
Representations of Pointers and References

- Pointers and References are single values stored in memory cells
- Intel microprocessors, addresses have two parts: a segment and an offset
 - Pointers and References are implemented in these systems as pairs of 16-bit cells, one for each of the two parts of an address

Solutions to the Dangling-Pointer Problem

- **Tombstones**

- Every heap-dynamic variable includes a special cell, called a tombstone, that is itself a pointer to the heap-dynamic variable
- Actual pointer variable points only at tombstones and never to heap-dynamic variables
- When a heap-dynamic variable is deallocated, the tombstone remains but is set to nil, indicating that the heap-dynamic variable no longer exists -> Approach prevents a pointer from ever pointing to a deallocated variable
- Any reference to any pointer that points to a nil tombstone can be detected as an error
- Costly in both time and space
- Their storage is never reclaimed
- Every access to a heap dynamic variable through a tombstone requires one more level of indirection

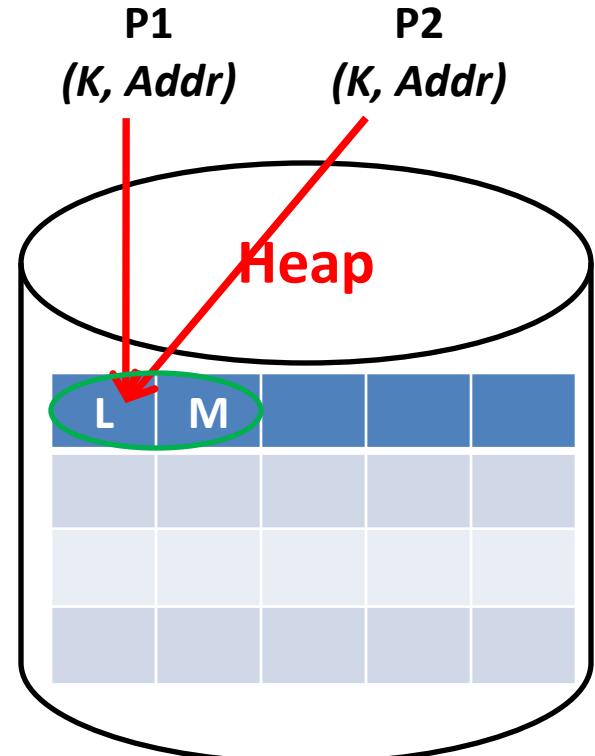


Solutions to the Dangling-Pointer Problem



- Lock-and-Keys Approach

- Pointer Values -> Ordered Pairs (Key, Address)
- Heap-dynamic variables -> Storage for the variable + a header cell (stores an integer lock value)
- When a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the heap-dynamic variable and in the key cell of the pointer that is specified in the call to new
- Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap-dynamic variable
- If they match, the access is legal
- Otherwise, the access is treated as a run-time error





Solutions to the Dangling-Pointer Problem

- Any copies of the pointer value to other pointers must copy the key value
- Therefore, any number of pointers can reference a given heap dynamic variable
- When a heap-dynamic variable is deallocated with dispose, its lock value is cleared to an illegal lock value
- Then, if a pointer other than the one specified in the dispose is dereferenced, its address value will still be intact, but its key value will no longer match the lock, so the access will not be allowed



Solutions to the Dangling-Pointer Problem

- Best solution to the dangling-pointer problem is to take deallocation of heap-dynamic variables out of the hands of programmers
- If programs cannot explicitly deallocate heap-dynamic variables, there will be no dangling pointers
- To do this, the run-time system must implicitly deallocate heap-dynamic variables when they are no longer useful

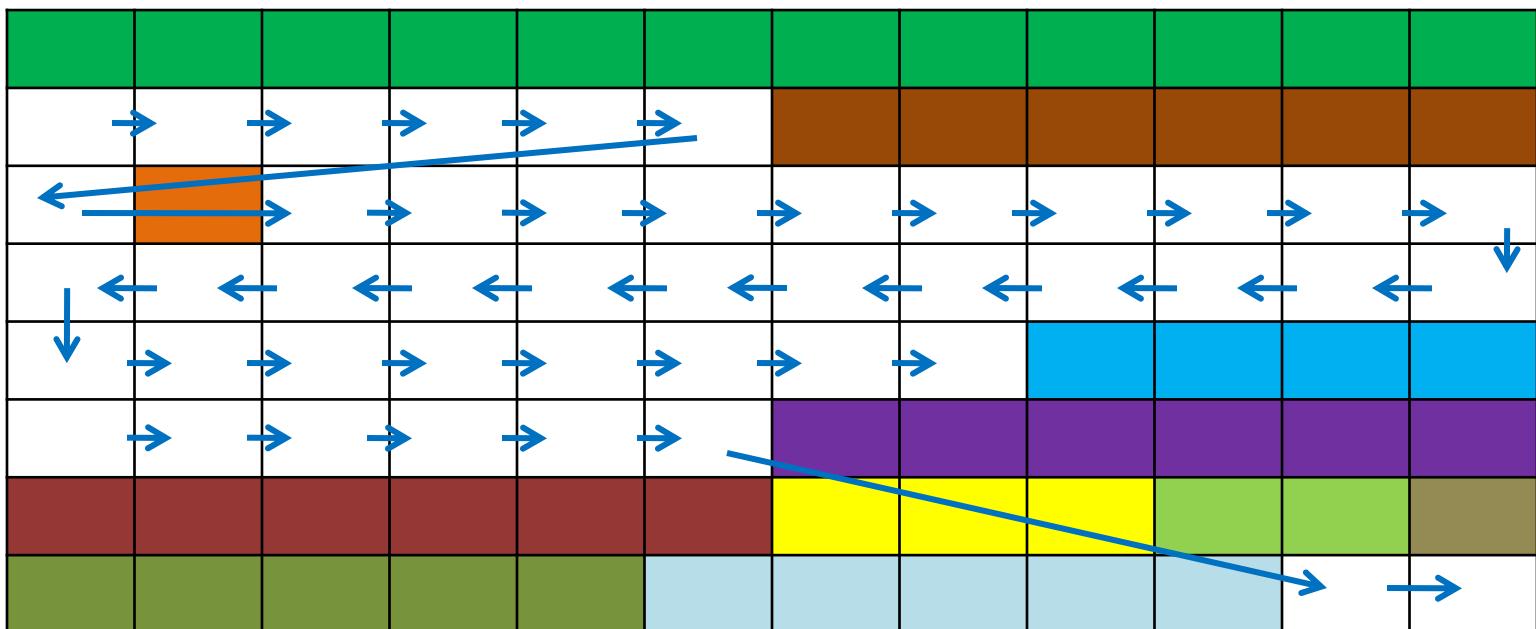


Heap Management

- Complex run-time process
- Examine in two separate situations
 - One in which all heap storage is allocated and deallocated in units of a single size
 - One in which variable-size segments are allocated and deallocated

Single-Size Cells

Heap Memory





Single-Size Cells

- In a single-size allocation heap, all available cells are linked together using the pointers in the cells, forming a list of available space
- Allocation is a simple matter of taking the required number of cells from this list when they are needed
- A heap-dynamic variable can be pointed to by more than one pointer, making it difficult to determine when the variable is no longer useful to the program
- Simply because one pointer is disconnected from a cell obviously does not make it garbage
- There could be several other pointers still pointing to the cell



Single-Size Cells

- Several different approaches to garbage collection
 - Reference Counters -> Reclamation is incremental and is done when inaccessible cells are created (Eager Approach)
 - Mark-Sweep -> Reclamation occurs only when the list of available space becomes empty (Lazy Approach)



Reference Counter Method

- Maintains in every cell a counter that stores the number of pointers that are currently pointing at the cell
- Embedded in the decrement operation for the reference counters, which occurs when a pointer is disconnected from the cell, is a check for a zero value
- If the reference counter reaches zero, it means that no program pointers are pointing at the cell, and it has thus become garbage and can be returned to the list of available space



Reference Counter Method

- **Problems**

- If storage cells are relatively small, the space required for the counters is significant
- Some execution time is obviously required to maintain the counter values
- Every time a pointer value is changed, the cell to which it was pointing must have its counter decremented, and the cell to which it is now pointing must have its counter incremented
 - Solution: Deferred Reference Counting -> Avoids reference counters for some pointers
- Complications arise when a collection of cells is connected circularly
 - Each cell in the circular list has a reference counter value of at least 1, which prevents it from being collected and placed back on the list of available space



Reference Counter Method

- **Advantages**
 - It is intrinsically incremental
 - Its actions are interleaved with those of the application, so it never causes significant delays in the execution of the application



Mark-and-Sweep Process

- Run-time system allocates storage cells as requested and disconnects pointers from cells as necessary, without regard for storage reclamation (allowing garbage to accumulate), until it has allocated all available cells
- At this point, a mark-sweep process is begun to gather all the garbage left floating around in the heap
- To facilitate the process, every heap cell has an extra indicator bit or field that is used by the collection algorithm



Mark-and-Sweep Process

- Consists of three distinct phases
 - First Phase -> All cells in the heap have their indicators set to indicate they are garbage
 - Marking Phase -> Every pointer in the program is traced into the heap and all reachable cells are marked as not being garbage
 - Sweep Phase -> All cells in the heap that have not been specifically marked as still being used, are returned to the list of available space

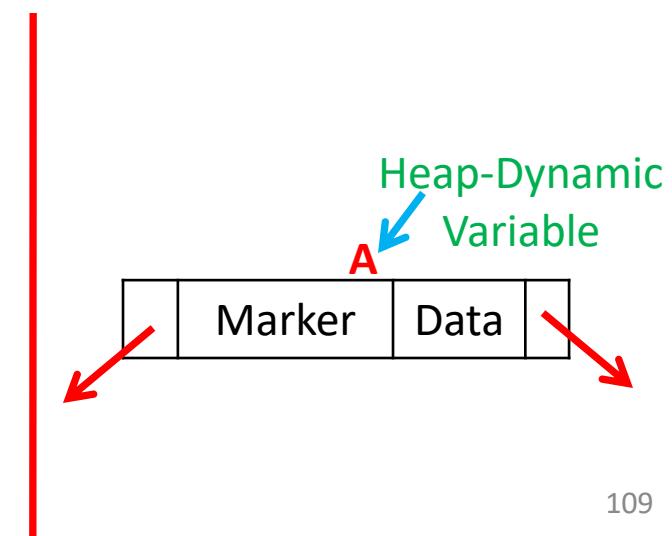


Mark-and-Sweep Process

- Assume that all heap-dynamic variables, or heap cells, consist of an information part; a part for the mark, named marker; and two pointers named llink and rlink
- These cells are used to build directed graphs with at most two edges leading from any node
- The marking algorithm traverses all spanning trees of the graphs, marking all cells that are found
- Like other graph traversals, the marking algorithm uses recursion

```
for every pointer r do
    mark(r)

void mark(void * ptr) {
    if (ptr != 0)
        if (*ptr.marker is not marked) {
            set *ptr.marker
            mark(*ptr.llink)
            mark(*ptr.rlink)
        }
}
```



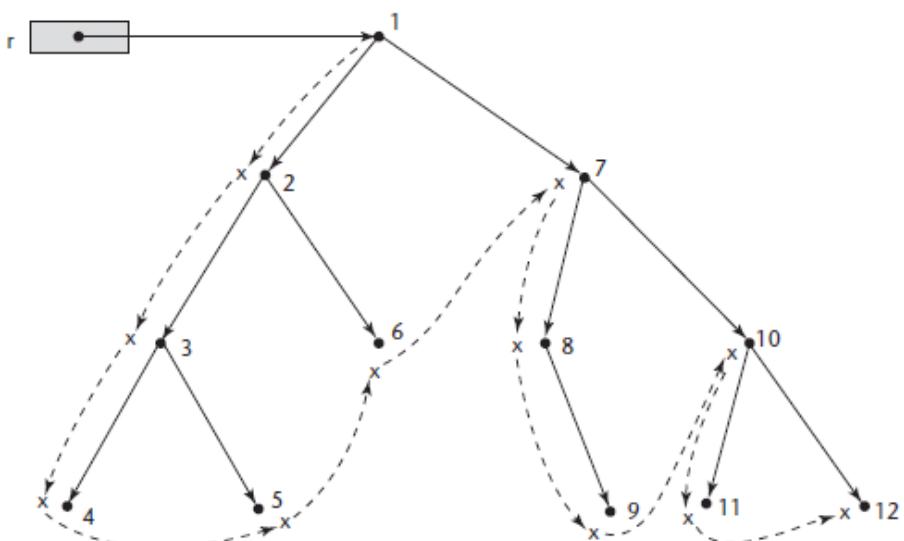
Mark-and-Sweep Process

```

for every pointer r do
    mark(r)

void mark(void * ptr) {
    if (ptr != 0)
        if (*ptr.marker is not marked) {
            set *ptr.marker
            mark(*ptr.llink)
            mark(*ptr.rlink)
        }
}

```



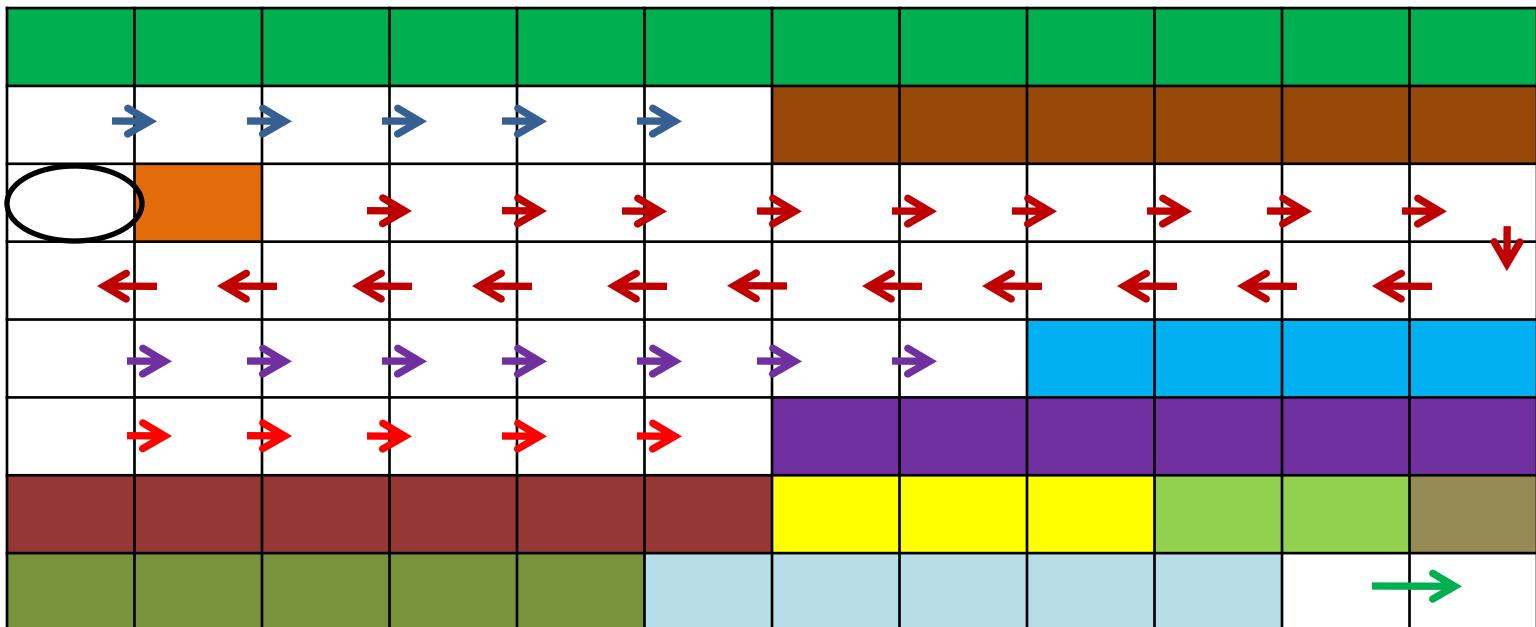


Mark-and-Sweep Process

- It was done too infrequently -> Only when a program had used all or nearly all of the heap storage
 - Takes a good deal of time
 - May yield only a small number of cells that can be placed on the list of available space
- Incremental Mark-Sweep -> Garbage collection occurs more frequently long before memory is exhausted
 - More effective in terms of the amount of storage that is reclaimed
 - Time required for each run of the process is obviously shorter, thus reducing the delay in application execution
- Perform the mark-sweep process on parts at different times

Variable-Size Cells

Heap Memory



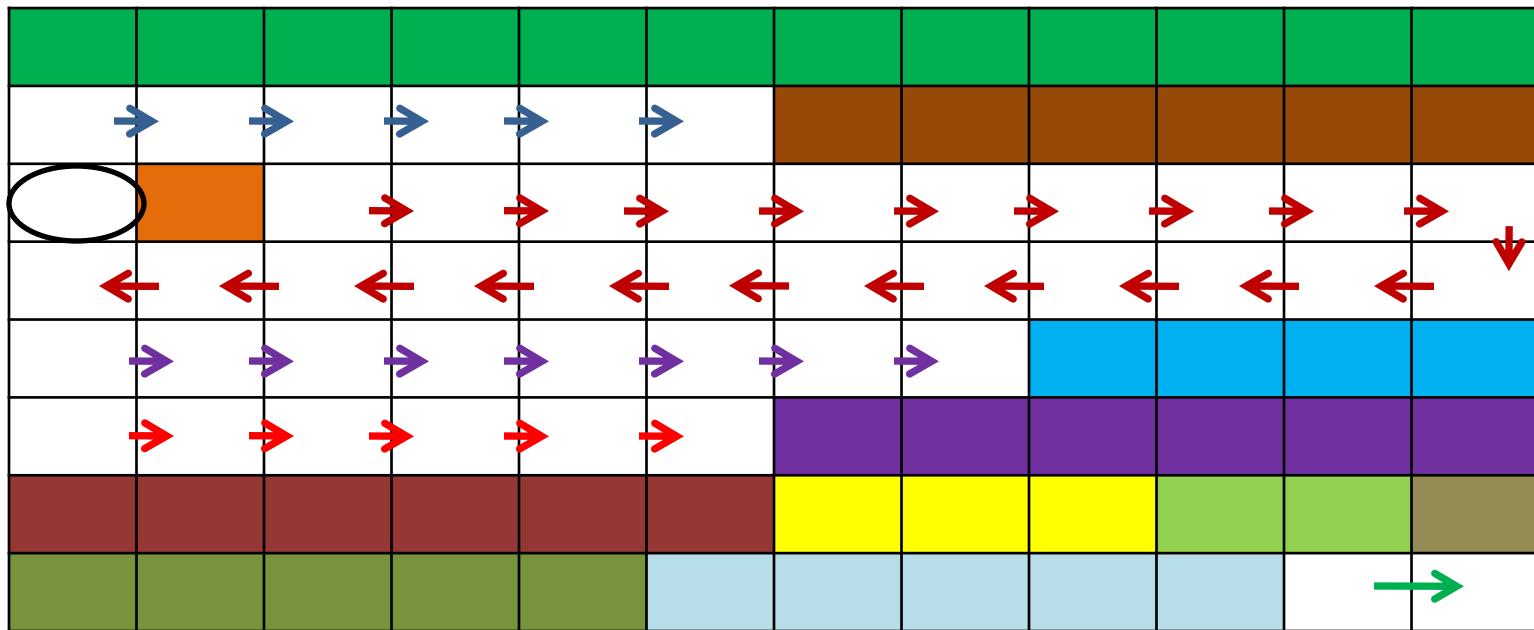


Variable-Size Cells

- Managing a heap from which variable-size cells are allocated has all the difficulties of managing one for single-size cells, but also has additional problems
- Additional problems posed by variable-size cell management depend on the method used
- If mark-sweep is used, the following additional problems occur
 - Initial setting of the indicators of all cells in the heap to indicate that they are garbage is difficult
 - Because the cells are different sizes, scanning them is a problem
 - One solution is to require each cell to have the cell size as its first field
 - Then the scanning can be done, although it takes slightly more space and somewhat more time than its counterpart for fixed-size cells

Variable-Sized Cells

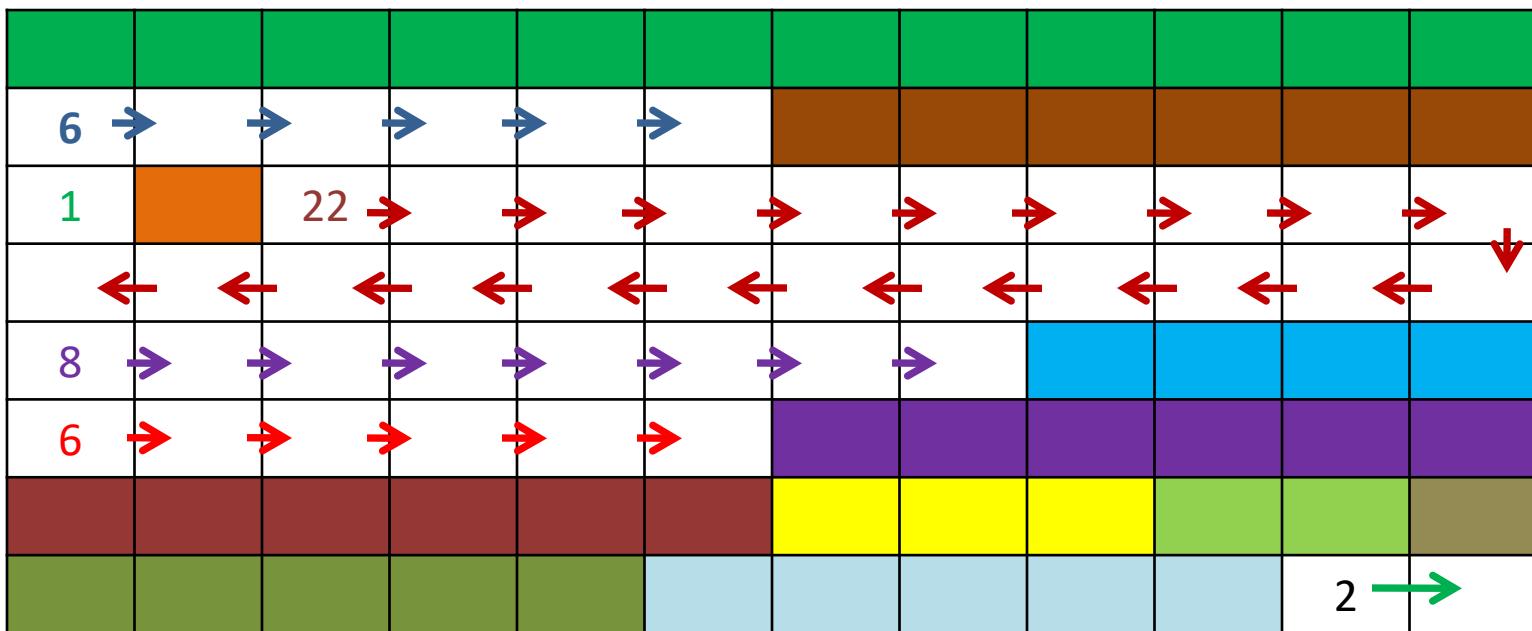
Heap Memory



Variable-Sized Cells

Heap Memory

First value of each available space is a header that represents the length of that available space





Variable-Size Cells

- Marking process is nontrivial
 - How can a chain be followed from a pointer if there is no predefined location for the pointer in the pointed-to-cell?
 - Cells that do not contain pointers at all are also a problem
 - Adding an internal pointer to each cell, which is maintained in the background by the run-time system, will work
 - However, this background maintenance processing adds both space and execution time overhead to the cost of running the program

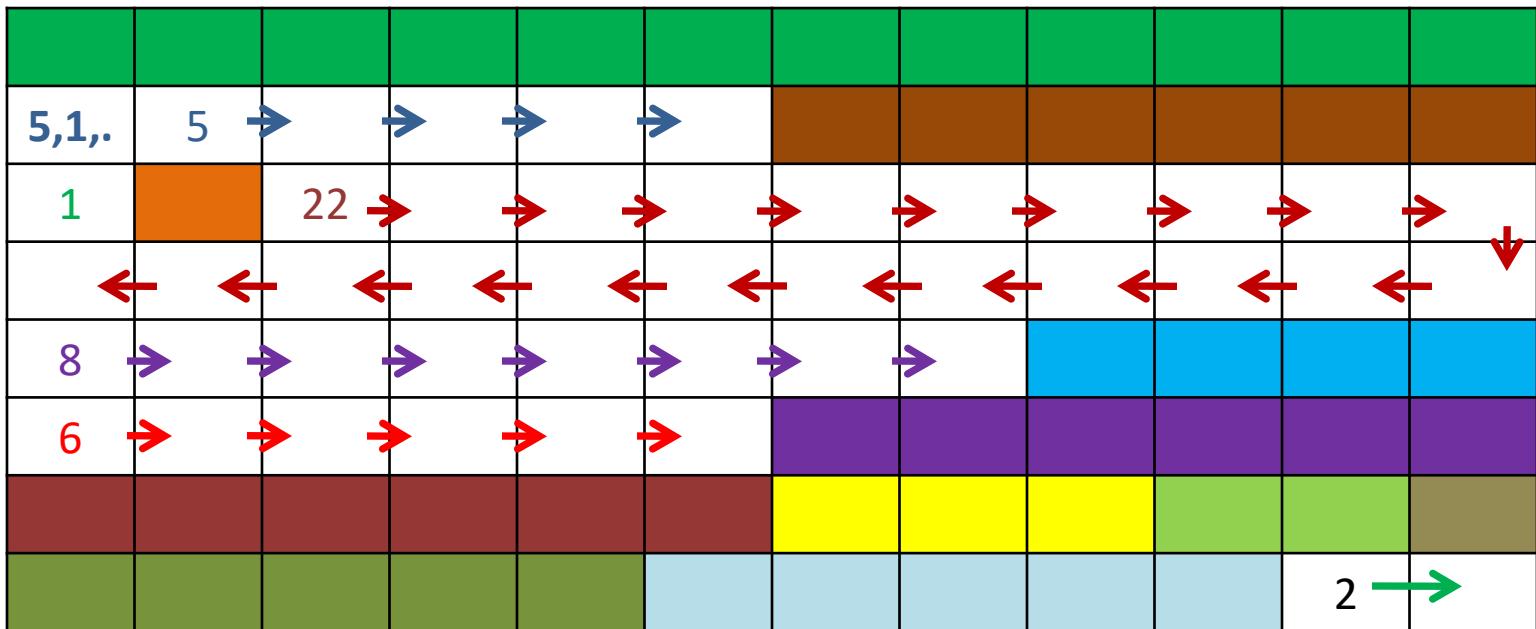


Variable-Size Cells

- Maintaining the list of available space is another source of overhead
 - The list can begin with a single cell consisting of all available space
 - Requests for segments simply reduce the size of this block
 - Reclaimed cells are added to the list
- The problem is that before long, the list becomes a long list of various-size segments, or blocks
- This slows allocation because requests cause the list to be searched for sufficiently large blocks
- Eventually, the list may consist of a large number of very small blocks, which are not large enough for most requests
- At this point, adjacent blocks may need to be collapsed into larger blocks
- Alternatives to using the first sufficiently large block on the list can shorten the search but require the list to be ordered by block size
- In either case, maintaining the list is additional overhead
- If reference counters are used, the first two problems are avoided, but the available-space list-maintenance problem remains

Variable-Sized Cells

Heap Memory





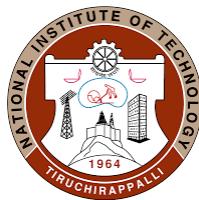
Miscellanoeus

Internal Fragmentation

- Difference between memory allocated and required space or memory is called Internal Fragmentation

External Fragmentation

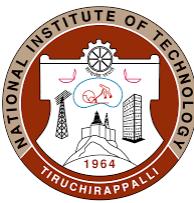
- Unused spaces formed between non-contiguous memory fragments are too small to serve a new request is called External fragmentation



CSPC31: Principles of Programming Languages

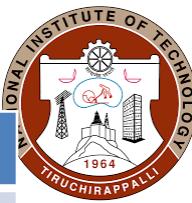
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 7 – Expressions and Assignment Statements



Objectives

- Semantics rules that determine the order of evaluation of operators in expressions
- Potential problem with operand evaluation order, when functions can have side effects
- Overloaded operators
- Mixed-mode expressions
- Relational and Boolean expressions
- Assignment statement



Introduction

- Fundamental means of specifying computations in a programming language
- Crucial for a programmer to understand both the syntax and semantics of expressions
- To understand expression evaluation, it is necessary to be familiar with the orders of operator and operand evaluation
- Operator evaluation order of expressions is dictated by the associativity and precedence rules
$$(A + B) + C = A + (B + C)$$
$$A + B * C \quad [* \uparrow \quad + \downarrow]$$
- Order of operand evaluation in expressions is often unstated by language designers
 - Allows implementors to choose the order, which leads to the possibility of programs producing different results in different implementations



Introduction

- Other issues -> Type Mismatches, Coercions, and Short-circuit evaluation
- Essence of imperative programming languages is the dominant role of assignment statements
- Purpose of these statements is to cause the side effect of changing the values of variables, or the state, of the program
 - Integral part of all imperative languages is the concept of variables whose values change during program execution
- Functional languages use variables of a different sort -> Parameters of functions
 - Also have declaration statements that bind values to names
 - These declarations are similar to assignment statements, but do not have side effects



Miscellaneous

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
int add(int, int);
int subtract(int, int);
void main()
{
    int a = 5, b = 10;
    int Summation = add(a, b);
}
```

```
int add(int x, int y)
{
    return subtract(x, y);
}
int subtract(int w, int v)
{
    return (w - v);
}
```



Arithmetic Expressions

- Arithmetic expressions consist of operators, operands, parentheses and function calls $(\textcolor{red}{a} + \textcolor{blue}{b}) + \textcolor{purple}{\text{fun}(c, d)}$
- An operator can be:
 - Unary -> Has single operand
 - Binary -> Has two operands
 - Ternary -> Has three operands
- In most programming languages, binary operators are infix, which means they appear between their operands $\textcolor{red}{a} + \textcolor{blue}{b}$
- Purpose of an arithmetic expression is to specify an arithmetic computation
- An implementation of such a computation must cause two actions -> fetching the operands + Executing arithmetic operations on those operands



Design Issues

- What are the operator precedence rules?
- What are the operator associativity rules?
- What is the order of operand evaluation?
- Are there restrictions on operand evaluation side effects?
- Does the language allow user-defined operator overloading?
- What type mixing is allowed in expressions?



Operator Evaluation Order

- Operator Precedence and Associativity Rules

Precedence

$$a + b * c \quad [a = 3; b = 4; c = 5; L \rightarrow R \Rightarrow 35; R \rightarrow L \Rightarrow 23]$$

- Mathematicians already defined -> Multiplication is considered to be of higher priority than addition, perhaps due to its higher level of complexity
- Operator precedence rules
 - Rules of the common imperative languages are nearly all the same
 - Exponentiation has the highest precedence, followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level
- Unary addition is called the identity operator -> Has no associated operation and thus has no effect on its operand
- Unary minus changes the sign of its operand

Operator Evaluation Order



- Java and C#, unary minus also causes the implicit conversion of short (16-bits) and byte (8-bits) operands to int (32-bits) type

$a + (- b) * c \Rightarrow a + (b -) * c \rightarrow \text{Legal}$

$a + - b * c \rightarrow \text{Illegal}$

- First two, relative precedence of the unary minus operator and the binary operator is irrelevant

$- a / b$

$- a * b$

$- a ** b$

- Exponentiation has higher precedence than unary minus

$- a ** b \Rightarrow - (a ** b)$

	Ruby	C-Based Languages
Highest	$**$	postfix $++, --$
	unary $+, -$	prefix $++, --, \text{unary } +, -$
	$, /, \%$	$, /, \%$
Lowest	binary $+, -$	binary $+, -$



Operator Evaluation Order

Associativity

$$a - b + c - d$$

- If addition and subtraction operators have the same level of precedence, the precedence rules say nothing about the order of evaluation of the operators
- Associativity Rules -> Left or Right associativity
- Associativity in common languages is left to right, except that the exponentiation operator sometimes associates right to left

$a - b + c$ -> Left operator is evaluated first

- Exponentiation in Fortran and Ruby is right associative

$$A ** B ** C$$

- Ada, exponentiation is nonassociative

$$(A ** B) ** C \Rightarrow A ** (B ** C)$$



Operator Evaluation Order

<i>Language</i>	<i>Associativity Rule</i>
Ruby	Left: *, /, +, - Right: **
C-based languages	Left: *, /, %, binary +, binary - Right: ++, --, unary -, unary +
Ada	Left: all except ** Nonassociative: **

- Many compilers for the common languages make use of the fact that some arithmetic operators are mathematically associative -> Arithmetic Addition
 $A + B + C + D \Rightarrow [1000000 + (-5900055) + 2000000 + (-499494949)]$
- A and C are very large positive numbers, and B and D are negative numbers with very large absolute values
- Adding B to A does not cause an overflow exception, but adding C to A does
- Likewise, adding C to B does not cause overflow, but adding D to B does
- Because of the limitations of computer arithmetic, addition is catastrophically nonassociative in this case
- Therefore, if the compiler reorders these addition operations, it affects the value of the expression



Operator Evaluation Order

Parantheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions
- Parenthesized part of an expression has precedence over its adjacent unparenthesized parts

$$(A + B) * C$$

- First operand of the multiplication operator is not available until the addition in the parenthesized sub expression is evaluated

$$A + B + C + D \Rightarrow (A + B) + (C + D) \quad // \text{Avoids Overflow}$$



Operator Evaluation Order

- Languages that allow parentheses in arithmetic expressions could dispense with all precedence rules and simply associate all operators left to right or right to left
- Programmer would specify the desired order of evaluation with parentheses
- Simple because neither the author nor the readers of programs would need to remember any precedence or associativity rules
- **Disadv:** Makes writing expressions more tedious, and it also seriously compromises the readability of the code



Operator Evaluation Order

Conditional Expressions

- if-then-else statements can be used to perform a conditional expression assignment

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

expression_1 ? expression_2 : expression_3

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

- where expression_1 is interpreted as a Boolean expression
- Question mark -> Beginning of the then clause
- Colon marks -> Beginning of the else clause
- Both clauses are mandatory
- ? is used in conditional expressions as a ternary operator



Operand Evaluation Order

- Variables in expressions are evaluated by fetching their values from memory
- Constants are sometimes evaluated the same way
- Constant may be part of the machine language instruction and not require a memory fetch $A + 80 \Rightarrow ADD A, \#80$
- If an operand is a parenthesized expression, all of the operators it contains, must be evaluated before its value can be used as an operand $(A + B - C * D) / F$
- If neither of the operands of an operator has side effects, then operand evaluation order is irrelevant
- Only interesting case arises when the evaluation of an operand does have side effects



Operand Evaluation Order

Side Effects

- Side effect of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable

$a + \text{fun}(a)$

- If fun does not have the side effect of changing a, then the order of evaluation of the two operands, a and fun(a), has no effect on the value of the expression
- However, if fun changes a, there is an effect

`a = 10;
b = a + fun(a);`

Eg: fun(a) returns value 10 as o/p and also changes the value of a to 20.

O/P: 20 (L -> R); 30 (R -> L)

```
int a = 5;
int fun1() {
    a = 17;
    return 3;
} /* end of fun1 */
void main() {
    a = a + fun1();
} /* end of main */
```

Operand Evaluation Order



- Two possible solutions to the problem -> Operand evaluation order and side effects
 - Language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects
 - Language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementors guarantee that order
- Disallowing functional side effects in the imperative languages is difficult, and it eliminates some flexibility for the programmer
- Consider the case of C and C++, which have only functions, meaning that all subprograms return one value
 - To eliminate the side effects of two-way parameters and still provide subprograms that return more than one value, the values would need to be placed in a struct and the struct returned
- Access to globals in functions would also have to be disallowed
- However, when efficiency is important, using access to global variables to avoid parameter passing is an important method of increasing execution speed
- In compilers, for example, global access to data such as the symbol table is commonplace



Operand Evaluation Order

- Problem with having a strict evaluation order is that some code optimization techniques used by compilers involve reordering operand evaluations
- A guaranteed order disallows those optimization methods when function calls are involved
- No perfect solution, as is borne out by actual language designs
- Java language definition guarantees that operands appear to be evaluated in left-to-right order, eliminating the problem



Operand Evaluation Order

Referential Transparency and Side Effects

- Concept of referential transparency is related to and affected by functional side effects
- A program has the property of referential transparency if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program
- Value of a referentially transparent function depends entirely on its parameters

```
→result1 = (fun(a) + b) / (fun(a) - c);  
      temp = fun(a);  
→result2 = (temp + b) / (temp - c);
```

- If *fun* changes the value of *a*, then that side effect violates the referential transparency of the program in which the code appears



Operand Evaluation Order

- Advantages to referentially transparent programs
 - Semantics of such programs is much easier to understand
 - Makes a function equivalent to a mathematical function, in terms of ease of understanding
- Because they do not have variables, programs written in pure functional languages are referentially transparent
- Functions in a pure functional language cannot have state, which would be stored in local variables
- If such a function uses a value from outside the function, that value must be a constant, since there are no variables
- Therefore, the value of the function depends on the values of its parameters



Overloaded Operators

- Arithmetic operators are often used for more than one purpose
 - “+” -> Addition, Subtraction, Concatenation
- Multiple use of an operator is called operator overloading and is generally thought to be acceptable, as long as neither readability nor reliability suffers
- “&” -> Binary Operator (Logical AND); Unary Operator (Address)
 $x = \&y;$
- Problems:
 - Using the same symbol for two completely unrelated operations is detrimental to readability
 - Simple keying error can go undetected
- “-” Operator



Overloaded Operators

- Some languages allow the programmer to further overload operator symbols
- Suppose a user wants to define the * operator between a scalar integer and an integer array

$a * b$ // $a = 5; b$ is an int array containing 5 elements

- + and * are overloaded for a matrix abstract data type and A, B, C, and D are variables of that type

$A * B + C * D \Rightarrow \text{MatrixAdd}(\text{MatrixMult}(A, B), \text{MatrixMult}(C, D))$

- C++ has a few operators that cannot be overloaded
 - Class or structure member operator (.) and the scope resolution operator (::)
- Operator overloading was one of the C++ features that was not copied into Java
- Reappear in C#



Type Conversions

- Type conversions are either narrowing or widening
- A narrowing conversion converts a value to a type that cannot store even approximations of all of the values of the original type. **Eg:** Double (64-bits) to Float (32-bits)
- A widening conversion converts a value to a type that can include at least approximations of all of the values of the original type. **Eg:** int (32-bit) to float (32-bit)
- Widening conversions are nearly always safe, meaning that the magnitude of the converted value is maintained
- Narrowing conversions are not always safe -> Sometimes the magnitude of the converted value is changed in the process.
Eg: Converting Float value 1.3E25 to Integer



Type Conversions

- Although widening conversions are usually safe, they can result in reduced accuracy
- In many language implementations, although integer-to-floating-point conversions are widening conversions, some precision may be lost
- Integers are stored in 32-bits, which allows at least nine decimal digits of precision

$$0 \text{ to } 2^{32} - 1 = 0 \text{ to } 429496729$$

- Floating-point values are also stored in 32 bits, with only about seven decimal digits of precision

0	-	1	0	0	0	0	0	0	-	1	0	0	1	0	0	1	0	0	0	1	1	1	1	1	1	0	1	1	0	1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0. **1415926**53589793 (*Obtained Value using Calc*)

Fraction Part of Pi = **0.1415925**02593994140625

- Integer-to-floating-point widening can result in the loss of two digits of precision



Type Conversions

Coercion in Expression

- Whether an operator can have operands of different types
- Languages that allow such expressions, which are called mixed-mode expressions, must define conventions for implicit operand type conversions because computers do not have binary operations that take operands of different types
- Coercion -> Implicit type conversion that is initiated by the compiler
- Type conversions explicitly requested by the programmer are referred to as explicit conversions, or casts, not coercions
- For overloaded operators in a language that uses static type binding, the compiler chooses the correct type of operation on the basis of the types of the operands



Type Conversions

- When the two operands of an operator are not of the same type and that is legal in the language, the compiler must choose one of them to be coerced and supply the code for that coercion
- Issues
 - Reliability problems -> Reduce the benefit of type checking
 - Whether programmers should be concerned with this category of errors or whether the compiler should detect them

`int a;`

`float b, c, d;`

`...`

`d = b * a; // "a" is a typo. "c" must be there`



Type Conversions

- Ada allows very few mixed type operands in expressions
- It does not allow mixing of integer and floating-point operands in an expression, with one exception
 - Exponentiation operator, `**`, can take either a floating-point or an integer type for the first operand and an integer type for the second operand
- C-based languages have integer types that are smaller than the int (32-bits) Type -> Byte (8-bits) and Short (16-Bits)
- Operands of all of these types are coerced to int whenever virtually any operator is applied to them
- Data can be stored in variables of these types, it cannot be manipulated before conversion to a larger type



Type Conversions

```
byte a, b, c;  
...  
a = b + c;
```

- Values of b and c are coerced to int and an int addition is performed
- Sum is converted to byte and put in “a”
- Given the large size of the memories of contemporary computers, there is little incentive to use byte and short, unless a large number of them must be stored



Type Conversions

Explicit Type Conversions

- Most languages provide some capability for doing explicit conversions, both widening and narrowing
- Warning messages are produced when an explicit narrowing conversion results in a significant change to the value of the object being converted
- Explicit type conversions are called casts
- To specify a cast, the desired type is placed in parentheses just before the expression to be converted

(int) angle

- Reason for having parantheses -> Two-word type names. Eg:
long int (at least 4 Bytes, usually 8 Bytes)



Type Conversions

Errors in Expressions

- Number of errors can occur during expression evaluation
- If the language requires type checking, either static or dynamic, then operand type errors cannot occur
- Other kinds of errors are due to the limitations of computer arithmetic and the inherent limitations of arithmetic
- Most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored
- Overflow or underflow, depending on whether the result was too large or too small
- One limitation of arithmetic is that division by zero is disallowed
- Floating-point overflow, underflow, and division by zero are examples of run-time errors -> Exceptions



Relational and Boolean Expressions

- Addition to Arithmetic Expressions, Programming Languages support Relational and Boolean Expressions

Relational Expressions

- Operator that compares the values of its two operands
- Relational expression has two operands and one relational operator. Eg: $a > b$
- Value of a relational expression is Boolean, except when Boolean is not a type included in the language
- Relational operators always have lower precedence than the arithmetic operators
 - $a + 1 > 2 * b \rightarrow$ Arithmetic expressions are evaluated first



Boolean Expressions

- Boolean expressions consist of Boolean variables, Boolean constants, relational expressions and Boolean operators
`const int true = 1`
`const int false = 0`
- Boolean Operators -> AND, OR, NOT operations, and sometime Exclusive OR and Equivalence
- Boolean operators usually take only Boolean operands (Boolean variables, Boolean literals or relational expressions) and produce Boolean values
- C-based languages assign a higher precedence to AND than OR (*AND -> Multiplication; OR -> Addition*)



Boolean Expressions

- Arithmetic expressions can be the operands of relational expressions

$$100 + 5 > 5 - 15$$

- Relational expressions can be the operands of Boolean expressions

$$a > b > c$$

- Three categories of operators must be placed in different precedence levels, relative to each other

<i>Highest</i>	postfix <code>++, --</code>
	unary <code>+, -, !</code> , prefix <code>++, --, !</code>
	<code>*, /, %</code>
	binary <code>+, -</code>
	<code><, >, <=, >=</code>
	<code>=, !=</code>
	<code>&&</code>
<i>Lowest</i>	<code> </code>



Boolean Expressions

0 -> False; 1 -> True

Perl and Ruby -> *&&* and *and* -> AND; *//* and *or* -> OR

- One difference between *&&* and *and* (*and* *//* and *or*) is that the spelled versions (*and* and *or*) have lower precedence
- Also, *and* and *or* have equal precedence, but *&&* has higher precedence than *//*
- C-based languages -> More than 40 non-arithmetic operators and at least 14 different levels of precedence
 - Evidence of the richness of the collections of operators and the complexity of expressions possible in these languages



Boolean Expressions

- Readability dictates that a language should include a Boolean type rather than simply using numeric types in Boolean expressions

bool b = True;

int b = 1;

- Some error detection is lost in the use of numeric types for Boolean operands because any numeric expression, whether intended or not, is a legal operand to a Boolean operator

int b = 10;

b + 5 > 5 + 10;

- In some imperative languages, any non-Boolean expression used as an operand of a Boolean operator is detected as an error



Short-Circuit Evaluation

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

$$(13 * a) * (b / 13 - 1) \quad // a = 0$$

- In arithmetic expressions, this shortcut is not easily detected during execution

$$(a >= 0) \&& (b < 10) \quad // a < 0$$

- This shortcut can be easily discovered during execution

```
index = 0;  
while ((index < listlen) && (list[index] != key))  
    index = index + 1;
```

- If evaluation is not short-circuit, and if key is not in list, the program will terminate with a subscript out-of-range exception



Short-Circuit Evaluation

```
listlen = len(list);  
index = 0;  
while ((index < listlen) && (list[index] != key))  
    index = index + 1;
```

- Suppose, if the number of iteration is equal to the listlen, then the condition (index < listlen) will fail
 - Suppose if the programming language facilitates short-circuit evaluation then the second condition (list[index] != key) will not be executed and hence the execution will successfully come out of the loop
 - However, if the programming language does not facilitate short-circuit evaluation, then the condition (list[index] != key) will try to access the element at position “index” in the array “list”. This throws the exception “out-of-range”



Short-Circuit Evaluation

($a > b$) || (($b++$) / 3)

- b is changed (in the second arithmetic expression) only when $a \leq b$
- If the programmer assumed b would be changed every time this expression is evaluated, during execution (and the program's correctness depends on it), the program will fail
- Ada:
 - Short-circuit operators -> *and then* and *or else*
 - Non-short-circuit operators -> *and* and *or*
- C:
 - Usual AND and OR operators, `&&` and `||`, respectively, are short-circuit
 - Bitwise AND and OR operators, `&` and `|`, respectively, that can be used on Boolean-valued operands and are not short-circuit



Assignment Statements

- Assignment statement is one of the central constructs in imperative languages
- Provides the mechanism by which the user can dynamically change the bindings of values to variables

int a = 5; // Static Binding

int a = b + 10 – 5 // Dynamic Binding

Simple Assignments

- Assignment Operator vs Equality Relational Operator

= or :=

vs

==

```
if ($flag) {  
    $count1 = 0;  
} else {  
    $count2 = 0;  
}
```

Conditional Targets

Perl ->

`($flag ? $count1 : $count2) = 0;`



Assignment Statements

Compound Assignment Operators

- Destination variable also appearing as the first operand in the expression on the right side

$a = a + b$

- Syntax of these assignment operators is the catenation of the desired binary operator to the = operator

$a += b$

Unary Assignment Operator

- Combine increment and decrement operations with assignment

$sum = ++ count;$

$sum = count ++;$

$count ++;$



Assignment Statements

- When two unary operators apply to the same operand, the association is right to left

- count ++ => - (count ++)

Assignment as an Expression

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

- Disadvantage of allowing assignment statements to be operands in expressions is that it provides yet another kind of expression side effect
- This type of side effect can lead to expressions that are difficult to read and understand

a = b + (c = d / b) - 1 =>

Assign d / b to c
Assign b + c to temp
Assign temp - 1 to a

- Multiple-target assignments, sum = count = 0;
- Loss of error detection in the C design of the assignment operation that frequently leads to program errors

if (x == y) ≠ if (x = y)



Assignment Statements

Multiple Assignments

- Multiple-target, multiple-source assignment statements
 $(\$first, \$second, \$third) = (20, 40, 60);$
- If the values of two variables must be interchanged
 $(\$first, \$second) = (\$second, \$first);$
- This correctly interchanges the values of \$first and \$second, without the use of a temporary variable (at least one created and managed by the programmer)



Mixed-Mode Assignment

- Frequently, assignment statements also are mixed mode
- Design question is: Does the type of the expression have to be the same as the type of the variable being assigned, or can coercion be used in some cases of type mismatch?

```
int a = 5, b = 10;  
float c = a / b;
```
- Fortran, C, C++, and Perl use coercion rules for mixed-mode assignment that are similar to those they use for mixed-mode expressions
 - Many of the possible type mixes are legal, with coercion freely applied
- C++, Java and C# allow mixed-mode assignment only if the required coercion is widening. **Eg:** int (32-bit) value can be assigned to a float (32-bit) variable, but not vice versa



Miscellaneous

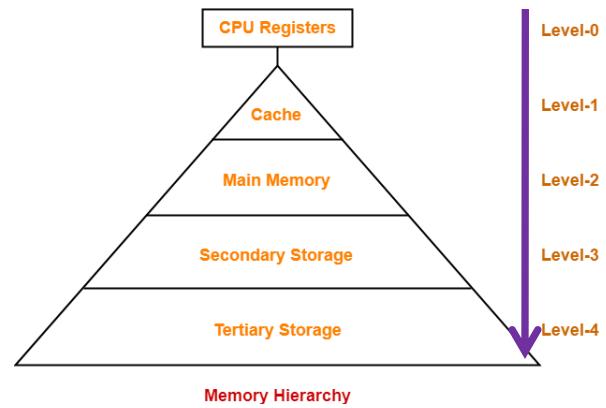
```
#include <stdio.h>
void main()
{
    int a = 5, b = 10;
    float c = a / b;
    printf("%f", c);
}
```

O/P: 0.000000

```
#include <stdio.h>
void main()
{
    int a = 5, b = 10;
    float c = (float) a / (float) b;
    printf("%f", c);
}
```

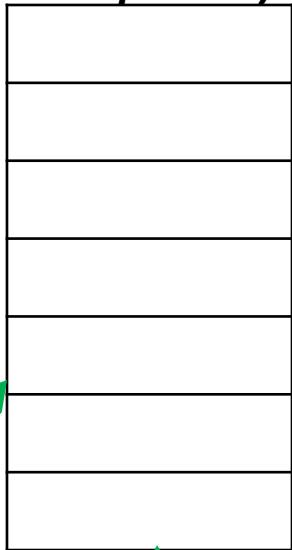
O/P: 0.500000

Miscellaneous



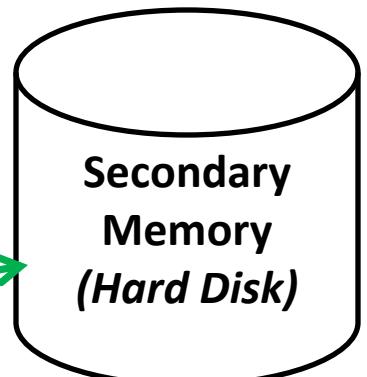
Cost-per-bit ↓
Access Time ↑

Main Memory
or RAM (DRAM
-> Capacitor)

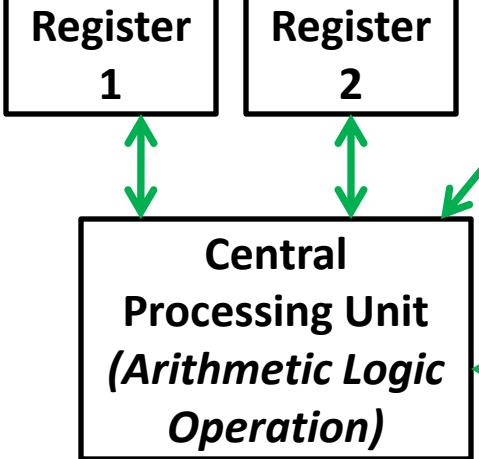


File Manager

Direct Memory Access

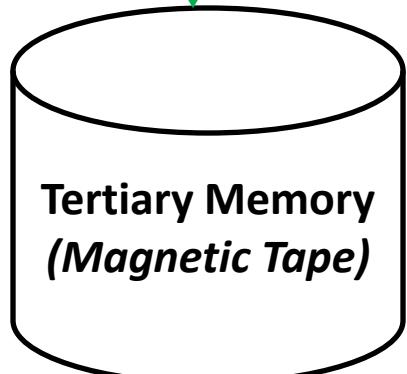


Secondary Memory
(Hard Disk)

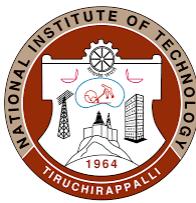


Cache Memory
(SRAM -> Flip Flop)

Internal Memory ->
Registers, Cache
Memory



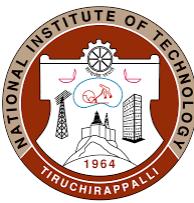
Tertiary Memory
(Magnetic Tape)



CSPC31: Principles of Programming Languages

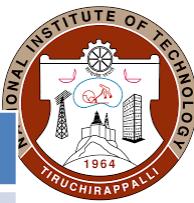
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 8 – Statement-Level Control Structures



Objectives

- Overview of the evolution of control statements
- Thorough examination of selection statements
- Variety of looping statements
- Problems associated with unconditional branch statements
- Guarded command control statements



Introduction

- Computations in imperative-language programs are accomplished by evaluating expressions and assigning the resulting values to variables
- Two additional linguistic mechanisms are necessary to make the computations in programs flexible and powerful
 - Some means of selecting among alternative control flow paths (of statement execution)
 - Some means of causing the repeated execution of statements or sequences of statements
- Statements that provide these kinds of capabilities are called control statements
- Great deal of research and discussion was devoted to control statements between the mid-1960s and the mid-1970s
- One of the primary conclusions of these efforts was that, although a single control statement (a selectable goto) is minimally sufficient, a language that is designed *not* to include a goto needs only a small number of different control statements



Introduction

- It was proven that all algorithms that can be expressed by flowcharts can be coded in a programming language with only two control statements
 - One for choosing between two control flow paths (*switch, if*)
 - One for logically controlled iterations (*for, while, do*)
- Programmers care less about the results of theoretical research on control statements than they do about writability and readability
- Rather than requiring the use of a logically controlled loop statement for all loops, it is easier to write programs when a counter-controlled loop statement can be used to build loops that are naturally controlled by a counter
- Primary factor that restricts the number of control statements in a language is readability
- Too few control statements can require the use of lower-level statements, such as the goto, which also makes programs less readable



Introduction

- How much a language should be expanded to increase its writability at the expense of its simplicity, size and readability
- A control structure is a control statement and the collection of statements whose execution it controls
- Only one design issue that is relevant to all of the selection and iteration control statements
 - Should the control structure have multiple entries?
- All selection and iteration constructs control the execution of code segments, and the question is whether the execution of those code segments always begins with the first statement in the segment
- It is now generally believed that multiple entries add little to the flexibility of a control statement, relative to the decrease in readability caused by the increased complexity
- Note that multiple entries are possible only in languages that include gotos and statement labels



Introduction

- Reader might wonder why multiple exits from control structures are not considered a design issue
- Reason is that all programming languages allow some form of multiple exits from control structures, the rationale being as follows
 - If all exits from a control structure are restricted to transferring control to the first statement following the structure, where control would flow if the control structure had no explicit exit, there is no harm to readability and also no danger
 - If an exit can have an unrestricted target and therefore can result in a transfer of control to anywhere in the program unit that contains the control structure, the harm to readability is the same as for a goto statement anywhere else in a program
- Languages that have a goto statement allow it to appear anywhere, including in a control structure
- Therefore, the issue is the inclusion of a goto, not whether multiple exits from control expressions are allowed



Miscellaneous

```
for(i = 0; i < n; i++)
{
    if (i > 10)
        break;
    else if (i > 5 && i < 10 && j = 0)
        break;
    else if (i > 0 and i < 5 && j =0)
        break;
    else
        i++;
}
int z =10; ←
```

```
# 60   for(i = 0; i < n; i++)
# 61   {
# 62       if (i > 10)
# 63           goto x; // x can be in line # 10, 110, 120, ...
# 64       else if (i > 5 && i < 10 && j = 0)
# 65           goto y; // y can be in line # 8, 1000, 1100, ...
# 66       else if (i > 0 and i < 5 && j =0)
# 67           goto z; // z can be in line # 18, 810, 820, ...
# 68   else
# 69       i++;
# 70   }
# 71   int z =10;
# 72   .
```



Selection Statements

- Selection statement provides the means of choosing between two or more execution paths in a program
- Selection statements fall into two general categories
 - Two-way
 - n-way or multiple selection

Two-way Selection Statements

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

Design Issues

- What is the form and type of the expression that controls the selection?
- How are the then and else clauses specified?
- How should the meaning of nested selectors be specified?



Selection Statements

Control Expression

- Control expressions are specified in parentheses if the then reserved word (or some other syntactic marker) is not used to introduce the then clause

bool c = true;

if (c) { ... } (or) if c then

if (x > 5) { ... } (or) if x > 5 then

Clause Form

- In many contemporary languages, the then and else clauses appear as either single statements or compound statements

if (x > 5)

{

....;

....;

}

if (x > 5):

....;

....;

printf()

if (x > 5)

....;

....;

end if;

if (x > 5)

....;

....;

fi;



Selection Statements

Nesting Selectors

```
<if_stmt> → if <logic_expr> then <stmt>
           | if <logic_expr> then <stmt> else <stmt>
```

- The issue was that when a selection statement is nested in the then clause of a selection statement, it is not clear to which if, an else clause should be associated

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

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

- Indentation seems to indicate that the else clause belongs with the first then clause
- In Java, as in many other imperative languages, the static semantics of the language specify that the else clause is always paired with the nearest previous unpaired then clause
- Static semantics rule, rather than a syntactic entity, is used to provide the disambiguation



Selection Statements

- C, C++, and C# have the same problem as Java with selection statement nesting
- Perl requires that all then and else clauses be compound -> Does not have this problem

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

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

```
if a > b then  
    sum = sum + a  
    account = account + 1  
else  
    sum = sum + b  
    bcount = bcount + 1  
end
```

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

- Use of a special word for this purpose resolves the question of the semantics of nested selectors and also adds to the readability of the statement
 - Design of the selection statement in Fortran 95+ Ada, Ruby, and Lua
 - Because the *end* reserved word closes the nested if, it is clear that the else clause is matched to the inner then clause



Selection Statements

Ruby

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

Python

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



Multiple Selection Constructs

- Multiple-selection statement allows the selection of one of any number of statements or statement groups -> Generalization of a selector
- Need to choose from among more than two control paths in a program is common
- Multiple selector can be built from two-way selectors and gotos
 - Resulting structures are cumbersome, unreliable, and difficult to write and read
- Need for a special structure is clear

```
switch (expression)
{
    case constant1:
        // statements
        break;

    case constant2:
        // statements
        break;

    .
    .
    .
    default:
        // default statements
}
```



Multiple Selection Constructs

Design Issues

- Type of expression on which the selector is based
- Whether single statements, compound statements, or statement sequences may be selected
- Whether only a single selectable segment (*case*) can be executed when the statement is executed
- Form of the case value specifications
- What should result from the selector expression evaluating to a value that does not select one of the segments
 - Such a value would be unrepresented among the selectable segments
 - Simply disallow the situation from arising and have the statement do nothing at all when it does arise



Multiple Selection Constructs

Summary of Design Issues

- What is the form and type of the expression that controls the selection?
- How are the selectable segments specified?
- Is execution flow through the structure restricted to include just a single selectable segment?
- How are the case values specified?
- How should unrepresented selector expression values be handled, if at all?



Multiple Selection Constructs

Examples of Multiple Selectors

```
switch (expression) {  
    case constant_expression1: statement1;  
    ...  
    case constantn: statementn;  
    [default: statementn+1]  
}
```

where the control expression and the constant expressions are some discrete type -> Integer Types, Character Types and Enumeration Types

- Selectable statements can be statement sequences, compound statements, or blocks
- Optional default segment is for unrepresented values of the control expression
- If the value of the control expression is not represented and no default segment is present, then the statement does nothing



Miscellaneous

```
0   1   2   3   4   5
enum coin{penny, nickel, dime, quarter, half-dollar, dollar};  
enum coin money;  
switch(money)  
{  
    case penny: printf("Penny");  
        break;  
    case nickel: printf("nickel");  
        break;  
}
```



Multiple Selection Constructs

- Switch statement does not provide implicit branches at the end of its code segment
- This allows control to flow through more than one selectable code segment on a single execution

```
switch (index) {  
    case 1:  
    case 3: odd += 1;  
              sumodd += index;  
    case 2:  
    case 4: even += 1;  
              sumeven += index;  
    default: printf("Error in switch, index = %d\n", index);  
}
```

This code prints the error message on every execution. Likewise, the code for the 2 and 4 constants is executed every time the code at the 1 or 3 constants is executed

- **break** statement, which is actually a restricted goto, is normally used for exiting switch statements



Multiple Selection Constructs

```
switch (index) {  
    case 1:  
    case 3: odd += 1;  
              sumodd += index;  
              break;  
    case 2:  
    case 4: even += 1;  
              sumeven += index;  
              break;  
    default: printf("Error in switch, index = %d\n", index);  
}
```

The segments for the case values 1 and 2 are empty, allowing control to flow to the segments for 3 and 4, respectively

- Occasionally, it is convenient to allow control to flow from one selectable code segment to another
- Reliability problem with this design arises when the mistaken absence of a break statement in a segment allows control to flow to the next segment incorrectly
- Designers of C's switch traded a decrease in reliability for an increase in flexibility



Multiple Selection Constructs

- C switch statement has virtually no restrictions on the placement of the case expressions, which are treated as if they were normal statement labels
- This laxness can result in highly complex structure within the switch body

```
switch (x)
    default:
        if (prime(x))
            case 2: case 3: case 5: case 7:
                process_prime(x);
        else
            case 4: case 6: case 8: case 9: case 10:
                process_composite(x);
```

Multiple Selection Constructs



```
switch (value) {      C#
    case -1:
        Negatives++;
        break;
    case 0:
        Zeros++;
        goto case 1;
    case 1:
        Positives++;
    default:
        Console.WriteLine("Error in switch \n");
}
```

- Rule is that every selectable segment must end with an explicit unconditional branch statement
 - `break` -> Transfers control out of the switch statement
 - `goto` -> Can transfer control to one of the selectable segments (or virtually anywhere else)
- Control expression and the case statements can be strings



Multiple Selection using if

- In many situations, a switch or case construct is inadequate for multiple selection
- **Eg:** When selections must be made on the basis of a boolean expression rather than some ordinal type, nested two-way selectors can be used to simulate a multiple selector
- Notice that this example is not easily simulated with a switch statement, because each selectable statement is chosen on the basis of a Boolean expression
- Else-if construct is not a redundant form of switch

```
if count < 10:  
    Bag1 = True;  
elif count < 100:  
    Bag2 = True;  
elif count < 1000:  
    Bag3 = True;
```



Iterative Statements

- Iterative statement is one that causes a statement or collection of statements to be executed zero, one, or more times
- An iterative statement is often called a loop
- If some means of repetitive execution of a statement or collection of statements were not possible, programmers would be required to state every action in sequence
- Useful programs would be huge and inflexible and take unacceptably large amounts of time to write and mammoth amounts of memory to store
- First iterative statements in programming languages were directly related to arrays
- Primary categories are defined by how designers answered two basic design questions
 - How is the iteration controlled?
 - Where should the control mechanism appear in the loop statement?



Iterative Statements

- Primary possibilities for iteration control are logical, counting, or a combination of the two
- Main choices for the location of the control mechanism are the top of the loop or the bottom of the loop
 - Issue is not the physical placement of the control mechanism
 - Rather, it is whether the mechanism is executed and affects control before or after execution of the statement's body
- Body of an iterative statement is the collection of statements whose execution is controlled by the iteration statement
- Pretest -> Test for loop completion occurs before the loop body is executed
- Posttest -> Occurs after the loop body is executed
- Iteration statement and the associated loop body together form an iteration statement



Iterative Statements

Counter Controlled Loops

- Counting iterative control statement has a variable, called the loop variable, in which the count value is maintained
- It also includes some means of specifying the initial and terminal values of the loop variable and the difference between sequential loop variable values, often called the stepsize. **Eg:** `for(int i = 0; i < 9; i++)`
 - Initial, Terminal and Step size specifications of a loop are called the loop parameters

Design Issues

- Type of the loop variable and that of the loop parameters obviously should be the same or at least compatible, but what types should be allowed?
 - One apparent choice is integer, but what about enumeration, character and floating-point types?



Iterative Statements

- Whether the loop variable is a normal variable, in terms of scope, or whether it should have some special scope
- Allowing the user to change the loop variable or the loop parameters within the loop can lead to code that is very difficult to understand, so another question is whether the additional flexibility that might be gained by allowing such changes is worth that additional complexity
- A similar question arises about the number of times and the specific time when the loop parameters are evaluated
 - If they are evaluated just once, it results in simple but less flexible loops



Iterative Statements

Summary of these 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, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?

The for Statement of the C-Based Languages



```
for (expression_1; expression_2; expression_3)
    loop body
```

- Loop body can be a single statement, a compound statement, or a null statement
- First expression is for initialization and is evaluated only once, when the for statement execution begins
- Second expression is the loop control and is evaluated before each execution of the loop body
 - Zero value means false and all nonzero values mean true
 - If the value of the second expression is zero, then for is terminated
 - Otherwise, the loop body statements are executed
- In C99, the expression also could be a Boolean type
- Last expression in the for is executed after each execution of the loop body
 - It is often used to increment the loop counter

The for Statement of the C-Based Languages



```
expression_1  
loop:  
  if expression_2 = 0 goto out  
  [loop body]  
  expression_3  
  goto loop  
out: . . .
```

```
for (count = 1; count <= 10; count++)  
  . . .  
}
```

- All of the expressions of C's for are optional
- An absent second expression is considered true, so a for without one is potentially an infinite loop
- If the first and/or third expressions are absent, no assumptions are made
- **Eg:** If the first expression is absent, it simply means that no initialization takes place

The for Statement of the C-Based Languages



- C for design choices are the following
 - There are no explicit loop variables or loop parameters
 - All involved variables can be changed in the loop body
 - Expressions are evaluated in the order stated previously

```
for (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum = ++count1 + count2, count2 *= 2.5);
```

```
count1 = 0
count2 = 1.0
loop:
    if count1 > 10 goto out
    if count2 > 100.0 goto out
    count1 = count1 + 1
    sum = count1 + count2
    count2 = count2 * 2.5
    goto loop
out: ...
```

- Whole expression is evaluated, but the resulting value is not used in the loop control

The for Statement of the C-Based Languages



- for statement of C99 and C++ differs from that of earlier versions of C in two ways
 - First, in addition to an arithmetic expression, it can use a Boolean expression for loop control
 - Second, the first expression can include variable definitions
- Scope of a variable defined in the for statement is from its definition to the end of the loop body
- In all of the C-based languages, the last two loop parameters are evaluated with every iteration
- Furthermore, variables that appear in the loop parameter expression can be changed in the loop body
- Therefore, these loops can be far more complex and are often less reliable than the counting loop of Ada



Logically Controlled Loops

- Collections of statements must be repeatedly executed, but the repetition control is based on a Boolean expression rather than a counter
- Logically controlled loop is convenient
- Logically controlled loops are more general than counter-controlled loops
- Every counting loop can be built with a logical loop, but the reverse is not true
- Recall that only selection and logical loops are essential to express the control structure of any flowchart

Design Issues

- Should the control be pretest or posttest?
- Should the logically controlled loop be a special form of a counting loop or a separate statement?



Logically Controlled Loops

Examples

- C-based programming languages include both pretest and posttest logically controlled loops that are not special forms of their counter-controlled iterative statements

```
while (control_expression)  
    loop body
```

```
do  
    loop body  
while (control_expression) ;
```

- In the pretest version of a logical loop (while), the statement or statement segment is executed as long as the expression evaluates to true
- In the posttest version (do), the loop body is executed until the expression evaluates to false
- Only real difference between the do and the while is that the do always causes the loop body to be executed at least once. In both cases, the statement can be compound



Logically Controlled Loops

while

loop:

```
if control_expression is false goto out  
[loop body]  
goto loop
```

out: . . .

do-while

loop:

```
[loop body]
```

```
if control_expression is true goto loop
```

- It is legal in both C and C++ to branch into both while and do loop bodies
- C89 version uses an arithmetic expression for control
- C99 and C++ -> It may be either arithmetic or Boolean



User-Located Loop Control

Mechanisms

- Sometimes, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop body
- Some languages provide this capability
- Such loops have the structure of infinite loops but include user-located loop exits
- Question is whether a single loop or several nested loops can be exited

Design Issues

- Should the conditional mechanism be an integral part of the exit?
- Should only one loop body be exited, or can enclosing loops also be exited?



User-Located Loop Control

Mechanisms

- C, C++, Python, Ruby, and C# have unconditional unlabeled exits (`break`)

```
outerLoop:  
    for (row = 0; row < numRows; row++)  
        for (col = 0; col < numCols; col++) {  
            sum += mat [row] [col];  
            if (sum > 1000.0)  
                break outerLoop;  
        }
```

```
while (sum < 1000) {  
    getnext (value);  
    if (value < 0) continue;  
    sum += value;  
}
```

- C, C++ and Python include an unlabeled control statement, `continue`, that transfers control to the control mechanism of the smallest enclosing loop
- This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop structure
- A negative value causes the assignment statement to be skipped, and control is transferred instead to the conditional at the top of the loop

User-Located Loop Control



Mechanisms

```
while (sum < 1000) {  
    getnext (value);  
    if (value < 0) break;  
    sum += value;  
}
```

- A negative value terminates the loop
- Break provide for multiple exits from loops, which may seem to be somewhat of a hindrance to readability
- However, unusual conditions that require loop termination are so common that such a statement is justified
- Furthermore, readability is not seriously harmed, because the target of all such loop exits is the first statement after the loop (or an enclosing loop) rather than just anywhere in the program
- The motivation for user-located loop exits is simple -> They fulfill a common need for goto statements through a highly restricted branch statement
- The target of a goto can be many places in the program, both above and below the goto itself
- However, the targets of user-located loop exits must be below the exit and can only follow immediately the end of a compound statement



Iteration Based on Data Structures

- A Do statement in Fortran uses a simple iterator over integer values
Do Count = 1, 9, 2
- Python -> for count in range [0, 9, 2]:
- A general data-based iteration statement uses a user-defined data structure and a user-defined function (the iterator) to go through the structure's elements
- Iterator is called at the beginning of each iteration, and each time it is called, the iterator returns an element from a particular data structure in some specific order
- **Eg:** Suppose a program has a user-defined binary tree of data nodes, and the data in each node must be processed in some particular order
- A user-defined iteration statement for the tree would successively set the loop variable to point to the nodes in the tree, one for each iteration



Iteration Based on Data Structures

- Initial execution of the user-defined iteration statement needs to issue a special call to the iterator to get the first tree element
- Iterator must always remember which node it presented last so that it visits all nodes without visiting any node more than once -> Iterator must be history sensitive
- A user-defined iteration statement terminates when the iterator fails to find more elements
- If the tree root is pointed to by a variable named root, and if traverse is a function that sets its parameter to point to the next element of a tree in the desired order

```
for (ptr = root; ptr == null; ptr = traverse(ptr)) {  
    ...  
}
```



Unconditional Branching

- Unconditional branch statement transfers execution control to a specified location in the program
- Unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements
- However, using the goto carelessly can lead to serious problems
- Goto has stunning power and great flexibility (all other control structures can be built with goto and a selector), but it is this power that makes its use dangerous
- Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain
- These problems follow directly from a goto's capability of forcing any program statement to follow any other in execution sequence, regardless of whether that statement precedes or follows the previously executed statement in textual order



Unconditional Branching

- Readability is best when the execution order of statements is nearly the same as the order in which they appear—in our case, this would mean top to bottom, which is the order with which we are accustomed
- Restricting gotos so they can transfer control only downward in a program partially alleviates the problem
- It allows gotos to transfer control around code sections in response to errors or unusual conditions but disallows their use to build any sort of loop
- A few languages have been designed without a goto. Eg: Java, Python, and Ruby
- Languages that have eliminated the goto have provided additional control statements, usually in the form of loop exits, to code one of the justifiable applications of the goto



Guarded Commands

- Methodology is described in Dijkstra (1976)
- Nondeterminism is sometimes needed in concurrent programs
- A selectable segment of a selection statement in a guarded-command statement can be considered independently of any other part of the statement, which is not true for the selection statements of the common programming languages
- Dijkstra's selection statement has the form

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

- Small blocks, called *fatbars*, are used to separate the guarded clauses and allow the clauses to be statement sequences
- Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence, is called a guarded command



Guarded Commands

- This selection statement has the appearance of a multiple selection, but its semantics is different
- All of the Boolean expressions are evaluated each time the statement is reached during execution
- If more than one expression is true, one of the corresponding statements can be nondeterministically chosen for execution

```
if i = 0 -> sum := sum + i  
[] i > j -> sum := sum + j  
[] j > i -> sum := sum + i  
fi
```

- If $i = 0 \&\& j > i$, then it chooses nondeterministically between the first and third assignment statements
- If i is equal to j and is not zero, a runtime error occurs because none of the conditions is true

- An implementation may always choose the statement associated with the first Boolean expression that evaluates to true
- But it may choose any statement associated with a true Boolean expression
- So, the correctness of the program cannot depend on which statement is chosen (among those associated with true Boolean expressions)
- If none of the Boolean expressions is true, a run-time error occurs that causes program termination
- This forces the programmer to consider and list all possibilities



Guarded Commands

- To find the largest of two numbers

```
if x >= y -> max := x  
[] y >= x -> max := y  
fi
```

- This computes the desired result without overspecifying the solution
- In particular, if x and y are equal, it does not matter which we assign to max
- This is a form of abstraction provided by the nondeterministic semantics of the statement

```
if (x >= y)  
    max = x;  
else  
    max = y;
```

```
if (x > y)  
    max = x;  
else  
    max = y;
```

- This choice between the two statements complicates the formal analysis of the code and the correctness proof of it
- This is one of the reasons why guarded commands were developed by Dijkstra



Guarded Commands

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

- If more than one is true, one of the associated statements is nondeterministically (perhaps randomly) chosen for execution, after which the expressions are again evaluated
- When all expressions are simultaneously false, the loop terminates
- Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that $q1 \leq q2 \leq q3 \leq q4$
- Without guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into the scalar variables q1, q2, q3, and q4

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```



Miscellaneous

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;  
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;  
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;  
od
```

- Let, q1 = 4; q2 = 3; q3 = 1; q4 = 2

Iteration 1

$q1 > q2 \rightarrow \text{True}$; $q2 > q3 \rightarrow \text{True}$; $q3 > q4 \rightarrow \text{False}$

- $q1 > q2 \rightarrow \text{temp} = 4; q1 = 3; q2 = 4$ // $q1 = 3; q2 = 4; q3 = 1; q4 = 2$

Iteration 2

$q1 > q2 \rightarrow \text{False}$; $q2 > q3 \rightarrow \text{True}$; $q3 > q4 \rightarrow \text{False}$

- $q2 > q3 \rightarrow \text{temp} = 4; q2 = 1; q3 = 4$ // $q1 = 3; q2 = 1; q3 = 4; q4 = 2$

Iteration 3

$q1 > q2 \rightarrow \text{True}$; $q2 > q3 \rightarrow \text{False}$; $q3 > q4 \rightarrow \text{True}$

- $q1 > q2 \rightarrow \text{temp} = 3; q1 = 1; q2 = 3$ // $q1 = 1; q2 = 3; q3 = 4; q4 = 2$

Iteration 4

$q1 > q2 \rightarrow \text{False}$; $q2 > q3 \rightarrow \text{False}$; $q3 > q4 \rightarrow \text{True}$

- $q3 > q4$ // $q1 = 1; q2 = 3; q3 = 2; q4 = 4$

Iteration 5

$q1 > q2 \rightarrow \text{False}$; $q2 > q3 \rightarrow \text{True}$; $q3 > q4 \rightarrow \text{False}$

- $q2 > q3$ // $q1 = 1; q2 = 2; q3 = 3; q4 = 4$

Iteration 6

- $q1 > q2 \rightarrow \text{False}$; $q2 > q3 \rightarrow \text{False}$; $q3 > q4 \rightarrow \text{False}$ // $q1 = 1; q2 = 2; q3 = 3; q4 = 4$

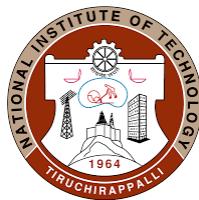


Guarded Commands

- Dijkstra's guarded command control statements are interesting, in part because they illustrate how the syntax and semantics of statements can have an impact on program verification and vice versa
- Program verification is virtually impossible when goto statements are used
- Verification is greatly simplified if
 - Only logical loops and selections are used
 - Only guarded commands are used
- It should be obvious, however, that there is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts



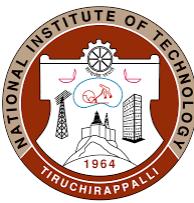
Thank You



CSPC31: Principles of Programming Languages

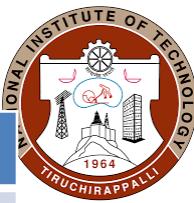
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 9 – Subprograms



Objectives

- Design of subprograms
 - Parameter passing methods
 - Local referencing environments
 - Overloaded subprograms
 - Generic subprograms
 - Aliasing
 - Side-effect problems



Miscellaneous

```
1000 #include<iostream.h>
1001 #include<conio.h>
1002 #include<stdlib.h>
1003 int add(int, int);
1004 int subtract(int, int);
1005 void main()
1006 {
1007     int a = 5, b = 10;
1008     int Summation = add(a, b);
1009 }
```

add(a, b); -> {1008, [a = 5, b = 10], [a, b]}

Subtract(a,b); -> {1014, [x = 5, y = 10], [x, y]}

```
1011 int add(int x, int y)
1012 {
1013     int res;
1014     res = subtract(x, y);
1015     return res;
1016 }
1017 int subtract(int w, int v)
1018 {
1019     int res;
1020     res = w - v;
1021     return res;
1022 }
```

Stack M/o

x, y
y = 10
x = 5
1014
a, b
b = 10
a = 5
1008

Store the Return Address, Values of Local Variables, Parameters in Stack Memory



Miscellaneous

```
1000 #include<iostream.h>
1001 #include<conio.h>
1002 #include<stdlib.h>
1003 int add(int, int);
1004 int subtract(int, int);
1005 void main()
1006 {
1007     int a = 5, b = 10;
1008     int Summation = add(a, b);
1009 }
```

add(a, b); -> {1008, [a = 5, b = 10], [a, b]}

Subtract(a,b); -> {1014, [x = 5, y = 10], [x, y]}

```
1011 int add(int x, int y)
1012 {
1013     int res;
1014     res = subtract(x, y);
1015     return res;
1016 }
1017 int subtract(int w, int v)
1018 {
1019     int res;
1020     res = w - v;
1021     return res;
1022 }
```

Stack M/o

v = 10

w = 5

Subtract(
x, y)

y = 10

x = 5

add(a, b)

b = 5

a = 5

main()

Store the Return Address, Values of Local
Variables, Parameters in Stack Memory



Introduction

- Two fundamental abstraction facilities -> Process Abstraction and Data Abstraction
- First programmable computer, Babbage's Analytical Engine (built in 1840s), had the capability of reusing collections of instruction cards at several different places in a program
- In a modern programming language, such a collection of statements is written as a subprogram
- This reuse results in several different kinds of savings -> Memory Space and Coding Time
- Instead of describing how some computation is to be done in a program, that description (the collection of statements in the subprogram) is enacted by a call statement, effectively abstracting away the details



Introduction

- This increases the readability of a program by emphasizing its logical structure while hiding the low-level details
- Methods of object-oriented languages are closely related to the subprograms
- Primary way methods differ from subprograms is the way they are called and their associations with classes and objects



Fundamentals of Subprogram

- Each subprogram has a single entry point
- Calling program unit is suspended during the execution of the called subprogram
 - Implies that there is only one subprogram in execution at any given time
- Control always returns to the caller when the subprogram execution terminates



Basic Definitions

- Subprogram Definition -> Describes the interface to and the actions of the subprogram abstraction
- Subprogram Call -> Explicit request that a specific subprogram be executed
- A subprogram is said to be active if, after having been called, it has begun execution but has not yet completed that execution
- Subprogram Header
 - First, it specifies that the following syntactic unit is a subprogram definition of some particular kind
 - Second, if the subprogram is not anonymous, the header provides a name for the subprogram
 - Third, it may optionally specify a list of parameters
void addr (parameters)



Basic Definitions

- Python -> **def** statements are executable
 - When a def statement is executed, it assigns the given name to the given function body
 - Until a function's def has been executed, the function cannot be called

```
if ...
    def fun( . . . ) :
        ...
else
    def fun( . . . ) :
        ...
        ...
```

- If the then clause of this selection construct is executed, that version of the function fun can be called, but not the version in the else clause
- If the else clause is chosen, its version of the function can be called but the one in the then clause



Basic Definitions

- All Lua functions are anonymous

```
function cube(x) return x * x * x end  
  
cube = function (x) return x * x * x end
```

- Parameter Profile of a subprogram contains the number, order and types of its formal parameters
- Protocol of a subprogram is its parameter profile plus, if it is a function, its return type
- Subprograms can have declarations as well as definitions



Parameters

- Subprograms typically describe computations
- Two ways that a non method subprogram can gain access to the data that it is to process
 - Through direct access to nonlocal variables (declared elsewhere but visible in the subprogram)
 - Through parameter passing
- Data passed through parameters are accessed through names that are local to the subprogram
- Parameter passing is more flexible than direct access to nonlocal variables
- Parameters in the subprogram header are called formal parameters
 - Dummy variables -> Not variables in the usual sense
 - Bound to storage only when the subprogram is called



Parameters

- Actual Parameters
- Binding of actual parameters to formal parameters is done by position -> Positional Parameters

`a = sum(a, b, c)`

`function sum (int x, int y, int z)`

- Keyword Parameters

```
sumer (length = my_length,  
       list = my_array,  
       sum = my_sum)
```

- Default value is used, if no actual parameter is passed to the formal parameter in the subprogram header

```
def compute_pay(income, exemptions = 1, tax_rate)
```

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```



Procedures and Functions

- Two distinct categories of subprograms -> Procedures and Functions
- Functions return values and procedures do not
- Procedures can produce results in the calling program unit by two methods:
 - If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them
 - If the procedure has formal parameters that allow the transfer of data to the caller, those parameters can be changed. **Eg:** Sorting of array elements



Procedures and Functions

- Functions structurally resemble procedures but are semantically modeled on mathematical functions
- If a function is a faithful model, it produces no side effects
 - It modifies neither its parameters nor any variables defined outside the function
- Functions define new user-defined operators

float power(float base, float exp)

result = 3.4 * power(10.0, x)



Design Issues for Subprograms

- Choice of one or more parameter-passing methods that will be used
- Whether the types of actual parameters will be type checked against the types of the corresponding formal parameters
- Whether local variables are statically or dynamically allocated
- Whether subprogram definitions can be nested
- Whether subprogram names can be passed as parameters
- Whether subprograms can be overloaded or generic



Design Issues for Subprograms

- Overloaded Subprogram -> Has the same name as another subprogram in the same referencing environment
- Generic Subprogram -> Computation can be done on data of different types in different calls
- Closure -> Nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program



Design Issues for Subprograms

- Are local variables statically or dynamically allocated?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter-passing method or methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- 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 subprograms be generic?
- If the language allows nested subprograms, are closures supported?



Local Referencing Environments

- Local Variables
- Subprograms can define their own variables, thereby defining local referencing environments
- Variables that are defined inside subprograms are called local variables
 - Scope is usually the body of the subprogram in which they are defined
- Local variables can be either static or stack dynamic
- If local variables are stack dynamic, they are bound to storage when the subprogram begins execution and are unbound from storage when that execution terminates
- **Adv of Stack Dynamic Local Variables:**
 - Flexibility they provide to the subprogram -> Recursive
 - Storage for local variables in an active subprogram can be shared with the local variables in all inactive subprograms



Local Variables

- **Disadv of Stack Dynamic Local Variable:**
 - Cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call to the subprogram
 - Accesses to stack-dynamic local variables must be indirect, whereas accesses to static variables can be direct -> Indirectness is required because the place in the stack where a particular local variable will reside can be determined only during execution
 - When all local variables are stack dynamic, subprograms cannot be history sensitive -> Cannot retain data values of local variables between calls



Local Variables

- History-sensitive subprograms
 - Generate pseudorandom numbers -> Store the last one in a static local variable
 - Coroutines and the subprograms used in iterator loop constructs
- Adv of static local variable:
 - More efficient—they require no run-time overhead for allocation and deallocation
 - If accessed directly, these accesses are obviously more efficient
 - Allow subprograms to be history sensitive
- Disadvantage of static local variables
 - Cannot support recursion
 - Their storage cannot be shared with the local variables of other inactive subprograms
- In most contemporary languages, local variables in a subprogram are by default stack dynamic
- In C and C++ functions, locals are stack dynamic unless specifically declared to be static



Local Variables

```
int adder(int list[], int listlen) {  
    static int sum = 0;  
    int count;  
    for (count = 0; count < listlen; count ++)  
        sum += list [count];  
    return sum;  
}
```

- **Python:**

- Any variable declared to be global in a method must be a variable defined outside the method
- A variable defined outside the method can be referenced in the method without declaring it to be global, but such a variable cannot be assigned in the method
- If the name of a global variable is assigned in a method, it is implicitly declared to be a local and the assignment does not disturb the global
- All local variables in Python methods are stack dynamic



Nested Subprograms

- Idea of nesting subprograms originated with Algol 60
- If a subprogram is needed only within another subprogram, why not place it there and hide it from the rest of the program?
- Descendants of Algol 60 -> Algol 68, Pascal and Ada
- Direct descendants of C -> Do not allow subprogram nesting
- JavaScript, Python, Ruby, and Lua -> Allow
- Most functional programming languages allow subprograms to be nested

```
function sub1() {  
    var x;  
    function sub2() {  
        alert(x); // Creates a dialog box with the value of x  
    };  
    function sub3() {  
        var x;  
        x = 3;  
        sub4(sub2);  
    };  
    function sub4(subx) {  
        var x;  
        x = 4;  
        subx();  
    };  
    x = 1;  
    sub3();  
};
```



Parameter Passing Methods

- Ways in which parameters are transmitted to and/or from called subprograms
- First, focus on the different semantics models of parameter-passing methods
- Then, discuss the various implementation models invented by language designers for these semantics models
- Next, survey the design choices of several languages and discuss the actual methods used to implement the implementation models
- Finally, consider the design considerations that face a language designer in choosing among the methods



Semantic Models of Parameter passing

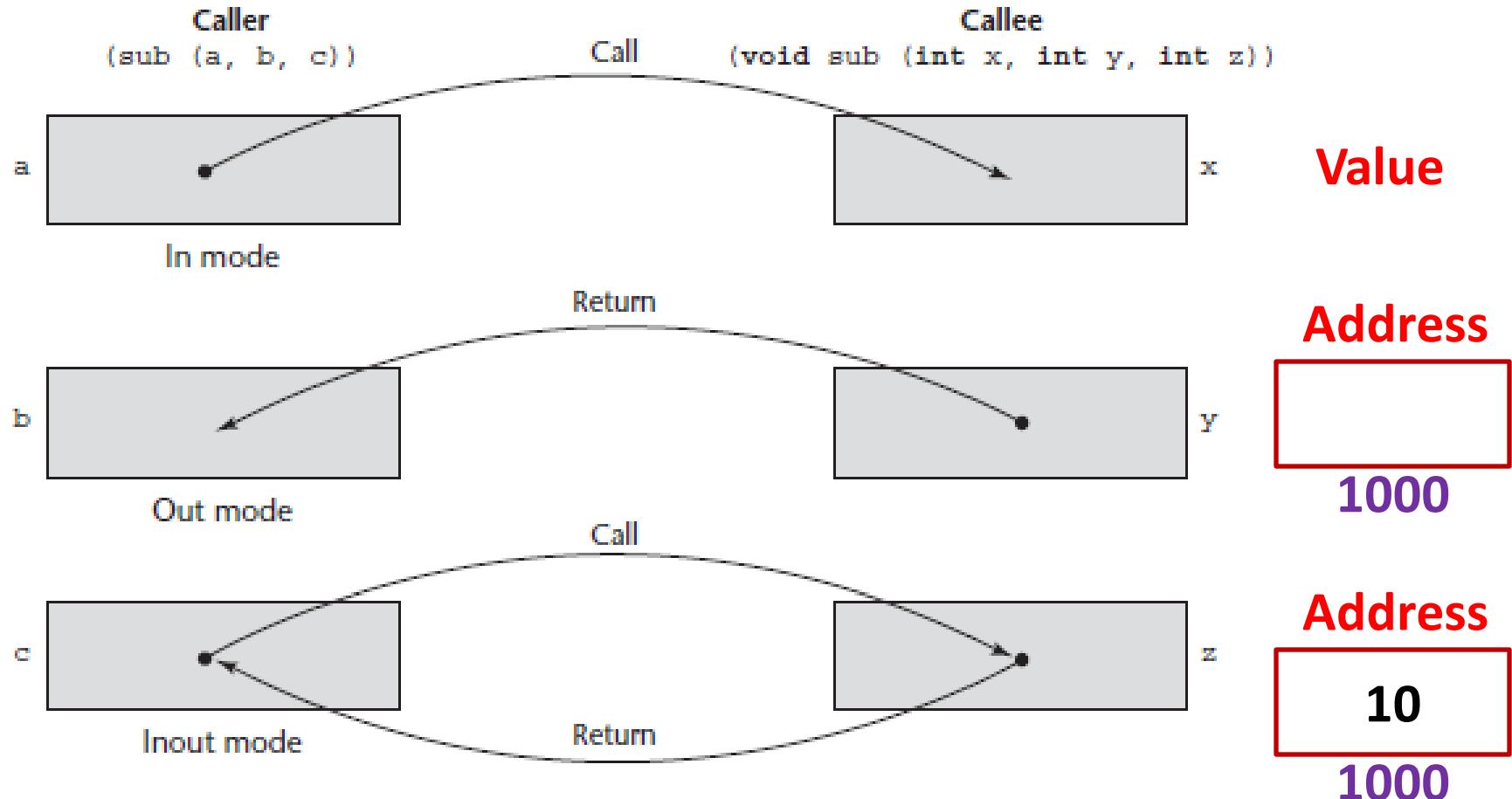
- Formal parameters are characterized by one of three distinct semantics models
 - Can receive data from the corresponding actual parameter (*in mode*)
 - Can transmit data to the actual parameter (*out mode*)
 - Can do both (*inout mode*)
- Eg: Two arrays of int values as parameters -> list1 and list2
 - Subprogram must add list1 to list2 and return the result as a revised version of list2
 - list1 (*in mode*); list2 (*inout mode*); Third array (*out mode*)
 - Third array should be *out mode*, because there is no initial value for this array and its computed value must be returned to the caller



Semantic Models of Parameter passing

- Two conceptual models of how data transfers take place in parameter transmission
 - An actual value is copied (to the caller, to the called, or both ways)
in mode *out mode*
inout mode
 - An access path is transmitted
- Most commonly, the access path is a simple pointer or reference

Implementation Models of Parameter Passing



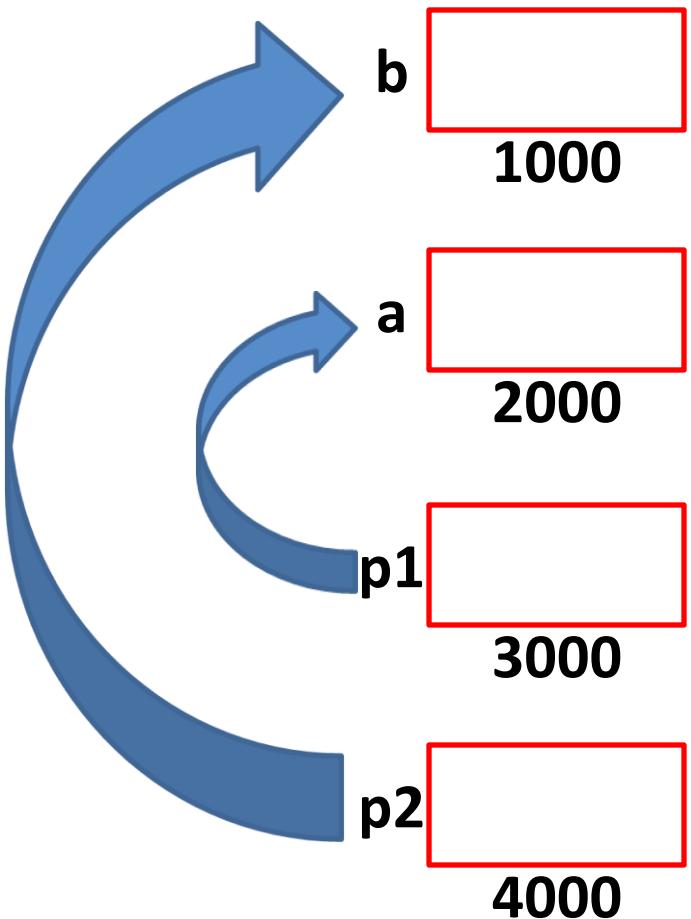


Pass-by-Value

- Value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram, thus implementing in-mode semantics
- Normally implemented by copy
- Could be implemented by transmitting an access path to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell
- **Adv:** For scalars it is fast, in both linkage cost and access time
- **Disadv:**
 - If copies are used, then an additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram
 - Actual parameter must be copied to the storage area for the corresponding formal parameter
- Storage and the copy operations can be costly if the parameter is large, such as an array with many elements

Pass-by-Value

```
void sub(int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(p1, p2);
```



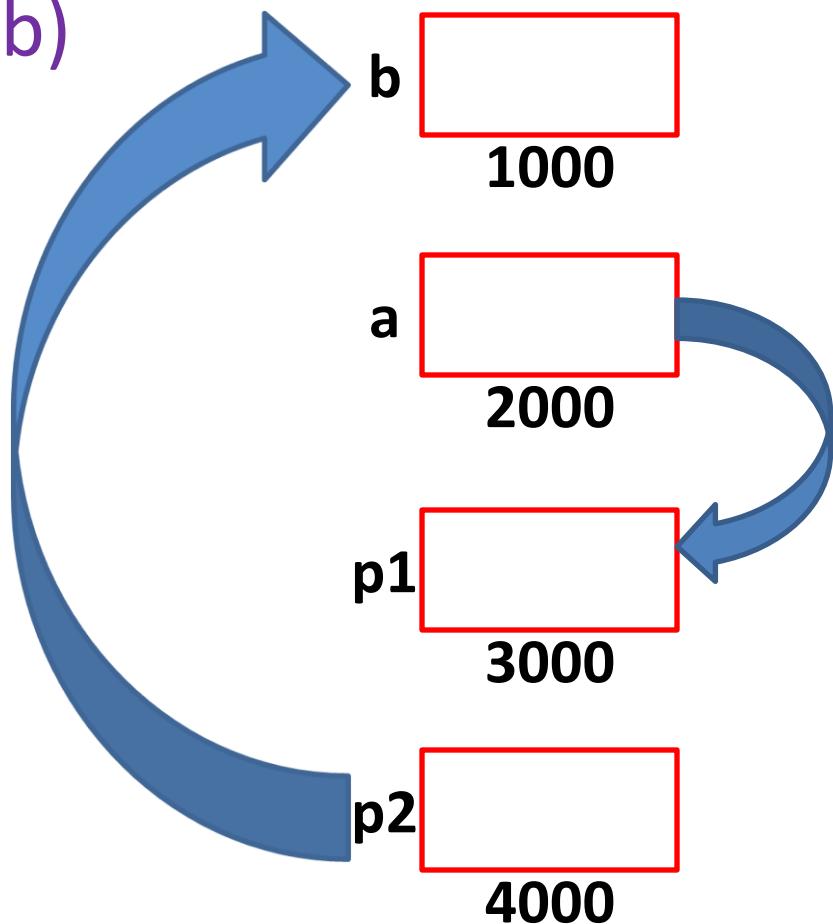


Pass-by-Result

- Implementation model for out-mode parameters
- When a parameter is passed-by-result, no value is transmitted to the subprogram
- The corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the caller's actual parameter, which obviously must be a variable
- Pass-by-result method has the advantages and disadvantages of pass-by-value, plus some additional disadvantages
- If values are returned by copy (as opposed to access paths), as they typically are, pass-by-result also requires the extra storage and the copy operations that are required by pass-by-value
- As with pass-by-value, the difficulty of implementing pass-by-result by transmitting an access path usually results in it being implemented by copy
 - In this case, the problem is in ensuring that the initial value of the actual parameter is not used in the called subprogram

Pass-by-Result

```
void sub(out int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(out p1, p2);
```



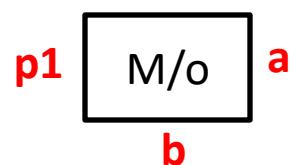


Pass-by-Result

- Additional Problem: Actual Parameter Collision

sub(p1, p1)

```
void sub(out int a, out int b)
{
    a = 17;
    b = 35;
}
int p1;
f.sub(out p1, out p1);
```

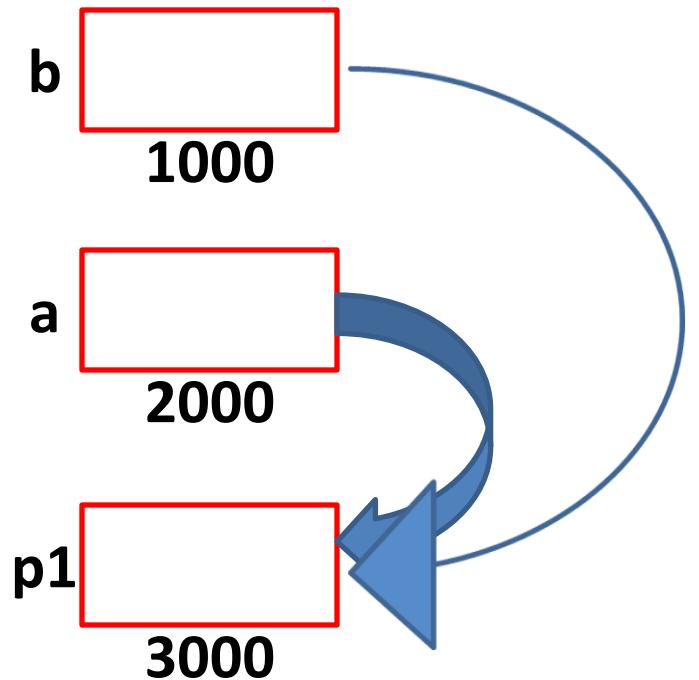


- If, at the end of the execution of sub, the formal parameter a is assigned to its corresponding actual parameter first, then the value of the actual parameter p1 in the caller will be 35 (***This happens when the parameters are evaluated from L -> R***)
- If b is assigned first, then the value of the actual parameter p1 in the caller will be 17 (***This happens when the parameters are evaluated from R -> L***)

- Order can be implementation dependent for some languages, different implementations can produce different results

Pass-by-Result

```
void sub(out int a, out int b)
{
    a = 17;
    b = 35;
}
int p1;
f.sub(out p1, out p1);
```





Pass-by-Result

- **Another Problem:** Implementor may be able to choose between two different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return

```
void DoIt(out int x, int index) {  
    x = 17;  
    index = 42;  
}  
....  
sub = 21;  
f.DoIt(list[sub], sub);
```

- If the address is computed on entry to the method, the value 17 will be returned to list[21]
- If computed just before return, 17 will be returned to list[42]

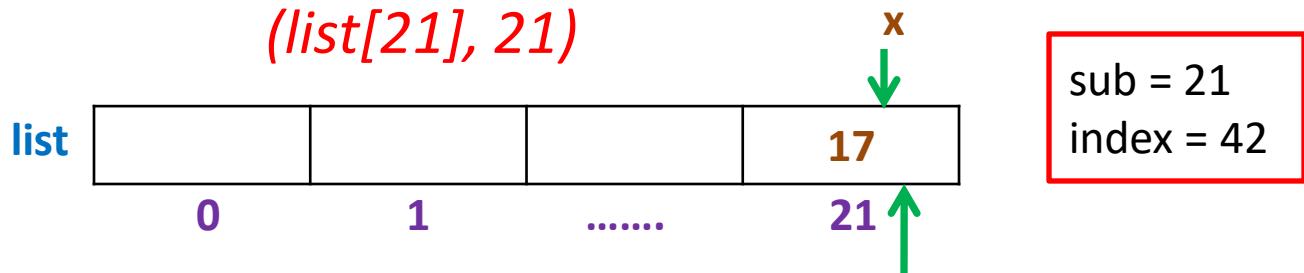
- Address of list[sub] changes between the beginning and end of the method
- Implementor must choose the time to bind this parameter to an address - at the time of the call or at the time of the return



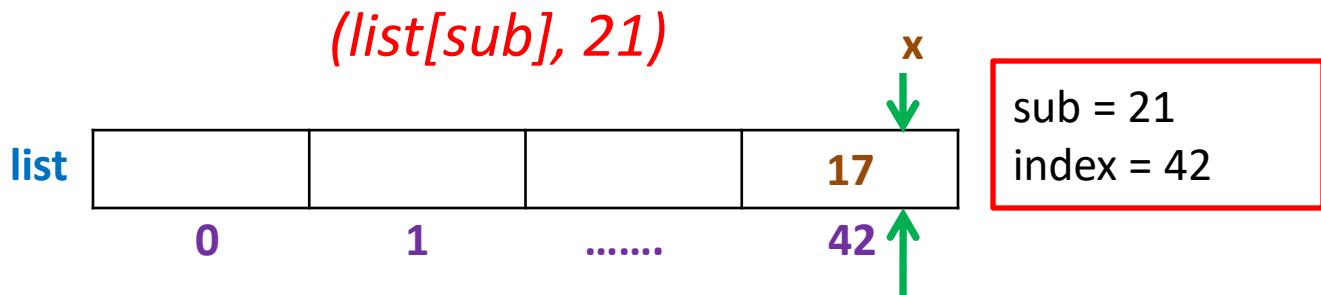
Pass-by-Result

```
void DoIt(out int x, int index) {  
    x = 17;  
    index = 42;  
}  
...  
sub = 21;  
f.DoIt(list[sub], sub);
```

- Address is computed on entry to the method:



- Address is computed just before return:



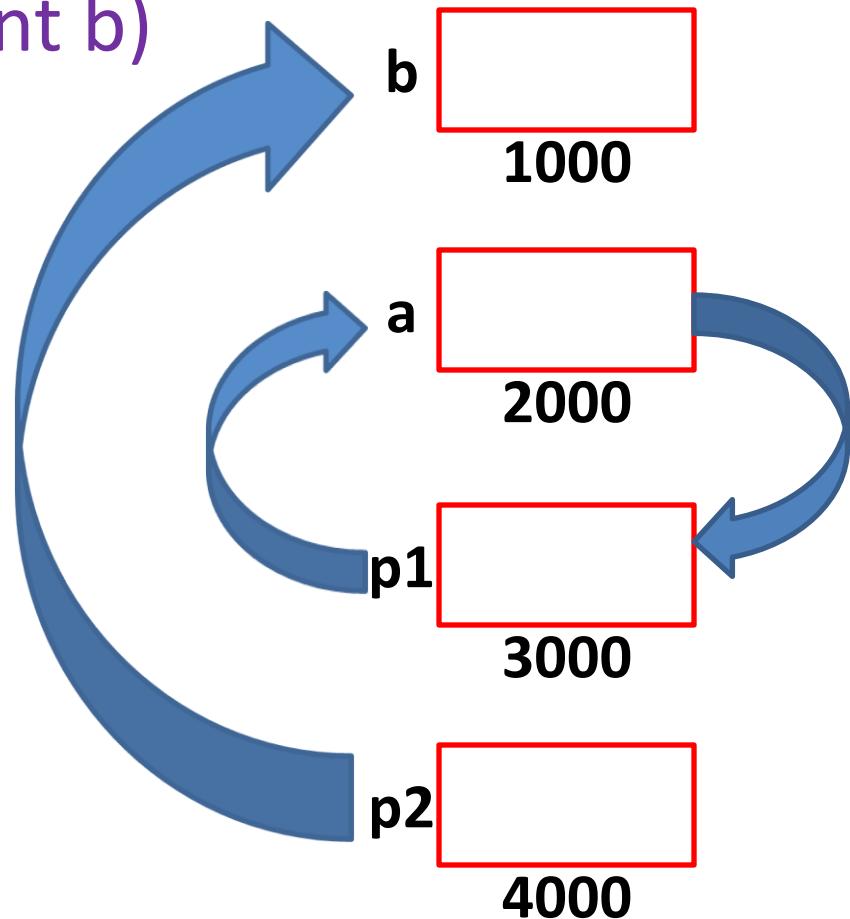


Pass-by-Result

- If the address is computed on entry to the method, the value 17 will be returned to list[21]
- If computed just before return, 17 will be returned to list[42]
- Makes programs unportable between an implementation that chooses to evaluate the addresses for out-mode parameters at the beginning of a subprogram and one that chooses to do that evaluation at the end

Pass-by-Value-Result

```
void sub(inout int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(inout p1, p2);
```



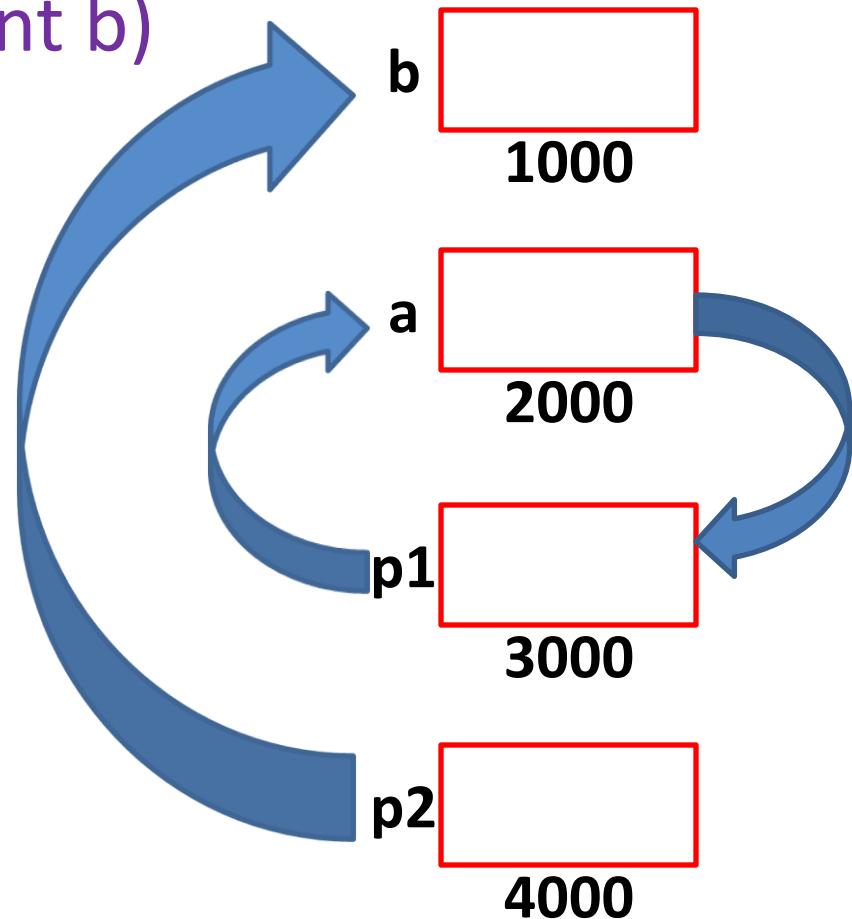


Pass-by-Value-Result

- Implementation model for inout-mode parameters in which actual values are copied
- Combination of both pass-by-value and pass-by-result
- The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable
- Pass-by-value-result formal parameters must have local storage associated with the called subprogram
- At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter
- Pass-by-value-result is sometimes called pass-by-copy, because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination

Pass-by-Value-Result

```
void sub(inout int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(inout p1, p2);
```





Pass-by-Value Result

- **Disadv:**
 - Requires multiple storage for parameters and time for copying values
 - Shares with pass-by-result the problems associated with the order in which actual parameters are assigned
- **Adv:** Relative to pass-by-reference

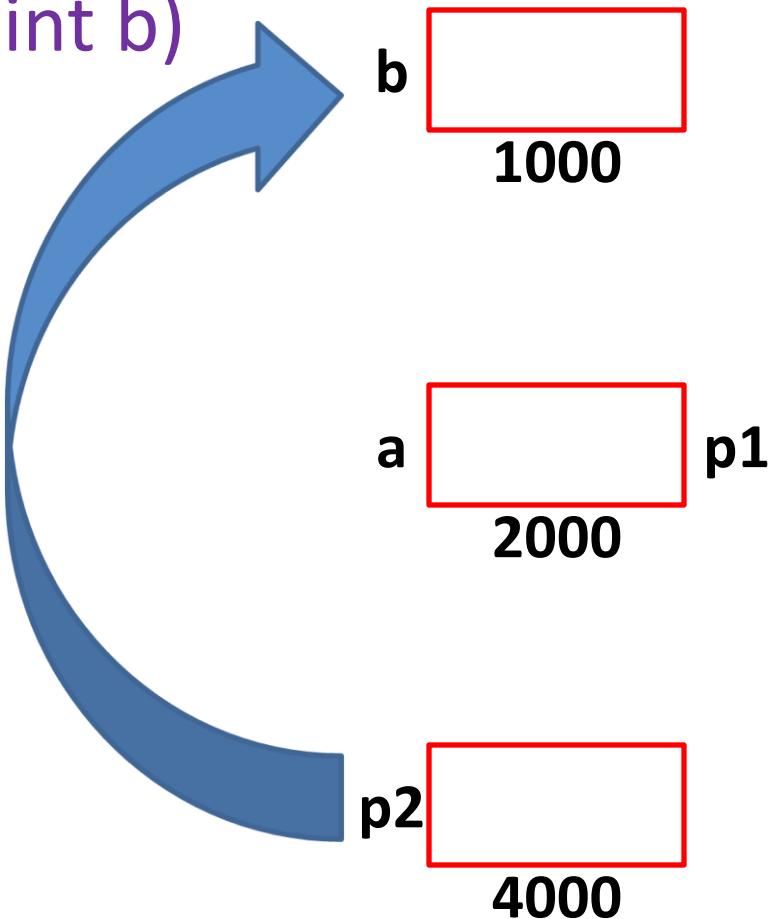


Pass-by-Reference

- Second implementation model for inout-mode parameters
- Pass-by-reference method transmits an access path, usually just an address, to the called subprogram
- This provides the access path to the cell storing the actual parameter
- Thus, the called subprogram is allowed to access the actual parameter in the calling program unit
- In effect, the actual parameter is shared with the called subprogram
- **Adv:** Duplicate space is not required, nor is any copying required

Pass-by-Reference

```
void sub(inout int *a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(inout &p1, p2);
```





Pass-by-Reference

- Disadv:
 - Access time
 - Not possible -> One-way communication to the called subprogram is required
 - Aliases can be created -> Harmful to Readability and Reliability
 - Collisions can occur



Pass-by-Reference

- Collisions can occur between actual parameters

void fun(int &first, int &second)

fun(total, total) → *first* and *second* are aliases

fun(list[i], list[j]) → If $i == j$, then *first* and *second* are aliases

fun1(list[i], list) → Aliasing in fun1

- Collisions can occur between formal parameters and nonlocal variables that are visible

- Inside sub -> *param* and *global* are aliases

```
int * global;
void main() {
    ...
    sub(global);
    ...
}

void sub(int * param) {
    ...
}
```



Pass-by-Name

- inout-mode parameter transmission method that does not correspond to a single implementation model
- When parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram
- A pass-by-name formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced
- Implementing a pass-by-name parameter requires a subprogram to be passed to the called subprogram to evaluate the address or value of the formal parameter



Pass-by-Name

- The referencing environment of the passed subprogram must also be passed
- This subprogram/referencing environment is a closure
- Pass-by-name parameters are both complex to implement and inefficient
- They also add significant complexity to the program, thereby lowering its readability and reliability

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))  
x = min(a, b);           → x = ((a) < (b) ? (a) : (b));  
y = min(1, 2);           → y = ((1) < (2) ? (1) : (2));  
z = min(a + 28, *p);    → z = ((a + 28) < (*p) ? (a + 28) : (*p));
```



Pass-by-Name

```
void sub(name int a, int b)
{
    b = a + 17;      // Replace a by p1 or 5
}

int p1 = 5;
int p2 = 10;
f.sub(name p1, p2);
```



Passing Values

```
void sub(int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(p1, p2);
```

1. Pass-by-Value (*in*)
2. Pass-by-Result (*out*)
3. Pass-by-Value-Result (*inout*)
4. Pass-by-Reference (*inout + [&, *]*)
5. Pass-by-Name (*name*)



Implementing Parameter Passing

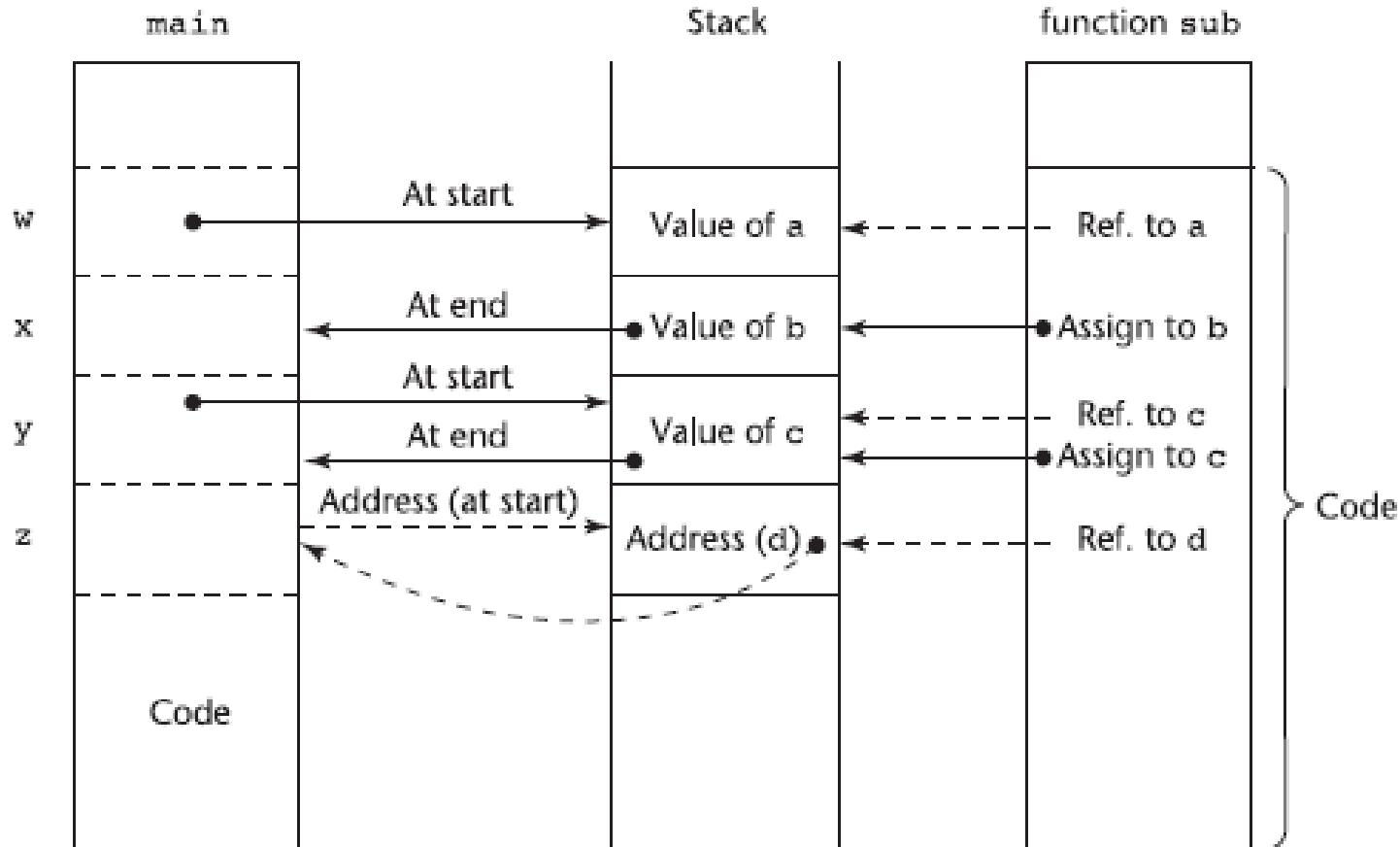
Methods

- Contemporary languages -> Parameter communication takes place through the run-time stack
- Run-time stack is initialized and maintained by the run-time system, which manages the execution of programs
- Runtime stack is used extensively for subprogram control linkage and parameter passing
- Following discussion assume that the stack is used for all parameter transmission

Pass-by-Value

- Parameters have their values copied into stack locations
- Stack locations then serve as storage for the corresponding formal parameters

Implementing Parameter Passing Methods



Function header: **void sub (int a, int b, int c, int d)**
 Function call in **main**: **sub (w,x,y,z)**
 (pass **w** by value, **x** by result, **y** by value-result, **z** by reference)



Implementing Parameter Passing Methods

Pass-by-Result

- Parameters are implemented as the opposite of pass-by-value
- Values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram

Pass-by-Value-Result

- Parameters can be implemented directly from their semantics as a combination of pass-by-value and pass-by-result
- Stack location for such a parameter is initialized by the call and is then used like a local variable in the called subprogram

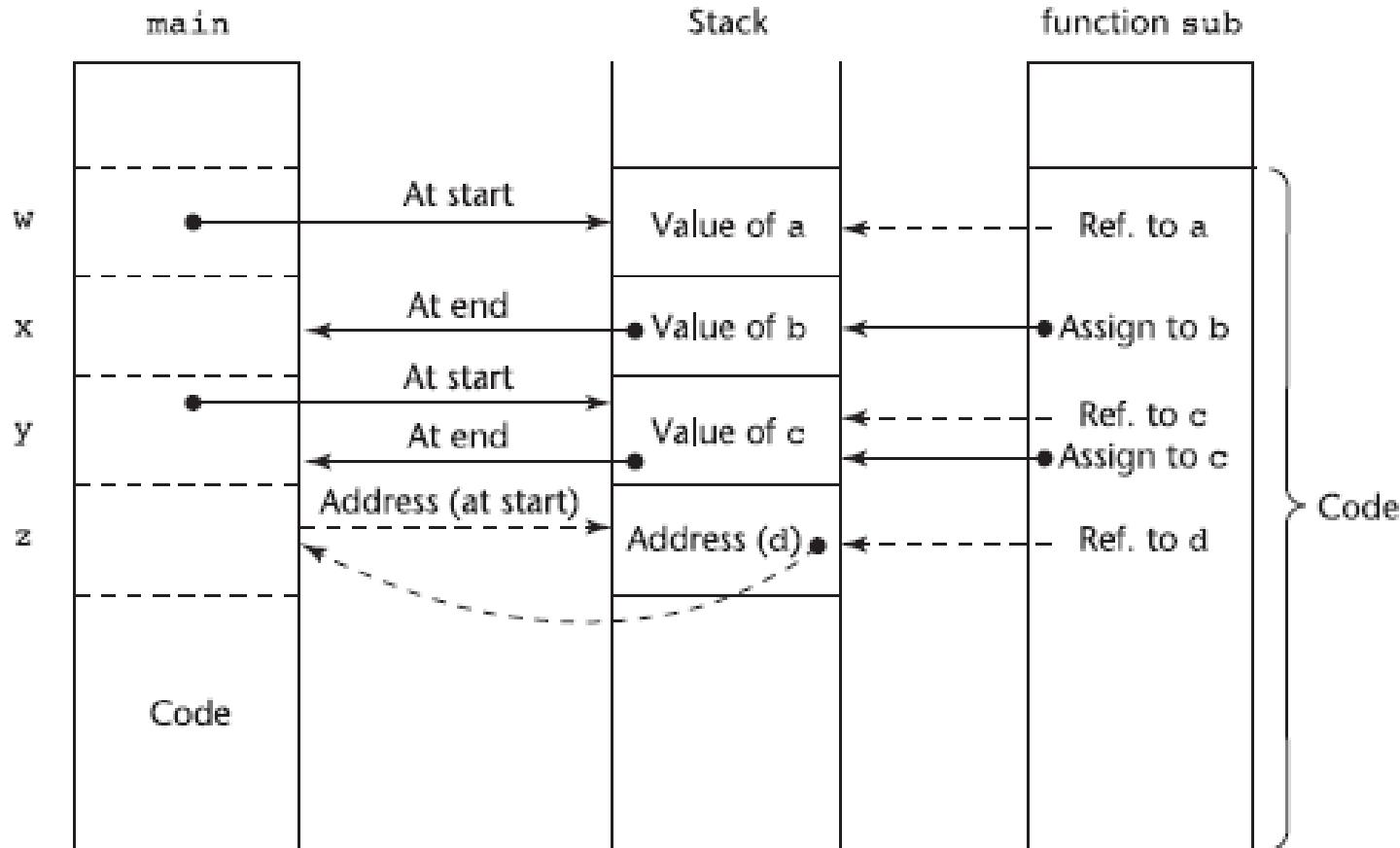


Implementing Parameter Passing Methods

Pass-by-Reference

- Simplest to implement
- Regardless of the type of the actual parameter, only its address must be placed in the stack
- In the case of literals, the address of the literal is put in the stack
- In the case of an expression, the compiler must build code to evaluate the expression just before the transfer of control to the called subprogram
- Address of the memory cell in which the code places the result of its evaluation is then put in the stack
- Compiler must be sure to prevent the called subprogram from changing parameters that are literals or expressions
- Access to the formal parameters in the called subprogram is by indirect addressing from the stack location of the address

Implementing Parameter Passing Methods



Function header: **void sub (int a, int b, int c, int d)**

Function call in **main**: **sub (w,x,y,z)**

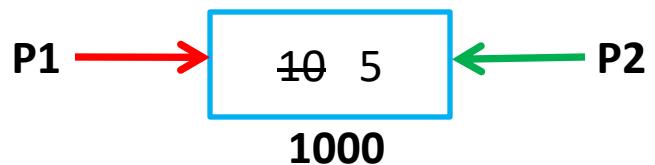
(pass **w** by value, **x** by result, **y** by value-result, **z** by reference)



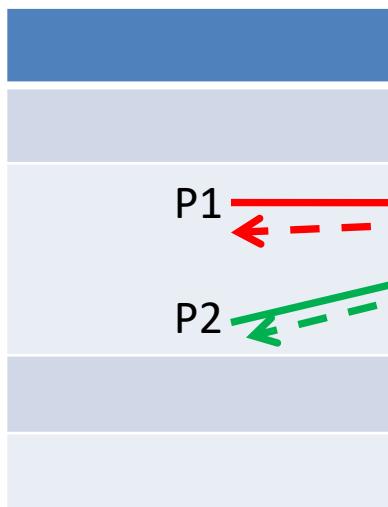
Implementing Parameter Passing Methods

- Error can occur with pass-by-reference and pass-by-value-result parameters, if care is not taken in their implementation
- Program contains two references to constant 10, the first as an actual parameter in a call to a subprogram
- Subprogram mistakenly changes the formal parameter that corresponds to the value 10 to the value 5
- Compiler for this program may have built a single location for the value 10 during compilation, as compilers often do, and uses that location for all references to the constant 10 in the program
- After the return from the subprogram, all subsequent occurrences of 10 will actually be references to the value 5
- Creates a programming problem that is very difficult to diagnose

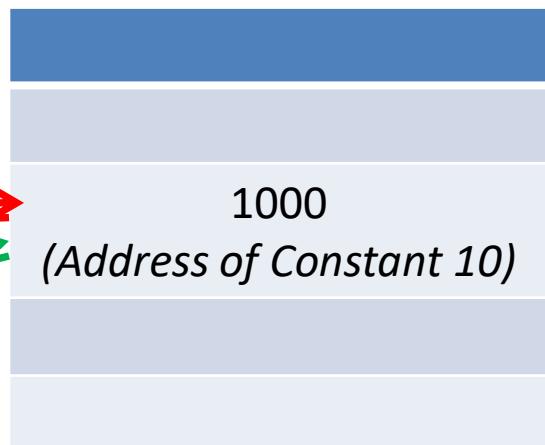
Implementing Parameter Passing Methods



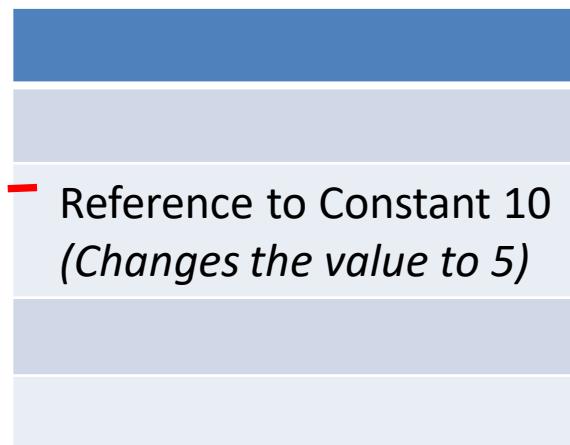
Main



Stack



Subprogram



Parameter Passing Methods of some Common Languages



- C uses pass-by-value
- Pass-by-reference (inout mode) semantics is achieved by using pointers as parameter
 - Value of the pointer is made available to the called function and nothing is copied back
 - However, because what was passed is an access path to the data of the caller, the called function can change the caller's data
- Formal parameters can be typed as pointers to constants
 - Corresponding actual parameters need not be constants, for in such cases they are coerced to constants
- Allows pointer parameters to provide the efficiency of pass-by-reference with the one-way semantics of pass-by-value
- Write protection of those parameters in the called function is implicitly specified



Miscellaneous

```
void sub(const int *a, int b)
{
    *a = 17; // Throws error
    b = *a + 35;
}
int p1 = 5;
int p2 = 10;
f.sub(&p1, p2);
```

Error: Assignment of Read-only Location



Parameter Passing Methods of some Common Languages

- C++ includes a special pointer type, called a *reference type* which is often used for parameters
- Reference parameters are implicitly dereferenced in the function or method, and their semantics is pass-by-reference
- C++ also allows reference parameters to be defined to be constants

```
void fun(const int &p1, int p2, int &p3) { . . . }
```

where p1 is pass-by-reference but cannot be changed in the function fun, p2 is pass-by-value, and p3 is pass-by-reference

- Neither p1 nor p3 need be explicitly dereferenced in fun
- Constant parameters and in-mode parameters are not exactly alike
- Constant parameters clearly implement in mode



Parameter Passing Methods of some Common Languages

- In a pure object-oriented language, the process of changing the value of a variable with an assignment statement, $x = x + 1$, does not change the object referenced by x
- Rather, it takes the object referenced by x , increments it by 1, thereby creating a new object (with the value $x + 1$), and then changes x to reference the new object
- So, when a reference to a scalar object is passed to a subprogram, the object being referenced cannot be changed in place
- Because the reference is passed by value, even though the formal parameter is changed in the subprogram, that change has no effect on the actual parameter in the caller
- Now, suppose a reference to an array is passed as a parameter
- If the corresponding formal parameter is assigned a new array object, there is no effect on the caller
- However, if the formal parameter is used to assign a value to an element of the array, as in $list[3] = 47$, the actual parameter is affected
- So, changing the reference of the formal parameter has no effect on the caller, but changing an element of the array that is passed as a parameter does



Type Checking Parameters

- Software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters
- Without such type checking, small typographical errors can lead to program errors that may be difficult to diagnose because they are not detected by the compiler or the run-time system

result = sub1(1)

the actual parameter is an integer constant

- If the formal parameter of sub1 is a floating-point type, no error will be detected without parameter type checking
- sub1 cannot produce a correct result given an integer actual parameter value when it expects a floating-point value



Type Checking Parameters

- C and C++ require some special discussion in the matter of parameter type checking
- In the original C, neither the number of parameters nor their types were checked
- In C89, the formal parameters of functions can be defined in two ways
 - Names of the parameters are listed in parentheses and the type declarations for them follow
 - Prototype method, in which the formal parameter types are included in the list

```
double sin(x)
double x;
{ ... }
```

```
double value;
int count;
...
value = sin(count);
```

```
double sin(double x)
{
    ...
}
value = sin(count);
```



Type Checking Parameters

```
int printf(const char* format_string, ...);
```

- A call to printf must include at least one parameter, a pointer to a literal character string

```
printf(); //Error: Too few arguments
```

```
printf("") //Warning: Zero-Length Format String
```

```
printf(" The String");
```

- Beyond that, anything (including nothing) is legal
- The way printf determines whether there are additional parameters is by the presence of format codes in the string parameter
- For example, the format code for integer output is %d
- This appears as part of the string, as in the following:

```
printf("The sum is %d\n", sum);
```

- The % tells the printf function that there is one more parameter



Multidimensional Arrays as Parameters

- Multidimensional arrays in C are really arrays of arrays, and they are stored in row major order

```
address (mat [i, j]) = address (mat [0, 0]) + i *  
                        number_of_columns + j
```

// Size of each element is 1

- Mapping function needs the number of columns but not the number of rows

```
void fun(int matrix[] [10]) {  
    ... }  
void main() {  
    int mat [5] [10];  
    ...  
    fun(mat);  
    ...  
}
```



Multidimensional Arrays as Parameters

- **Pbm:** Does not allow a programmer to write a function that can accept matrixes with different numbers of columns
- Matrix can be passed as a pointer, and the actual dimensions of the matrix also can be passed as parameters
- Then, the function can evaluate the user-written storage-mapping function using pointer arithmetic each time an element of the matrix must be referenced

```
void fun(float *mat_ptr,  
        int num_rows,  
        int num_cols);
```

```
* (mat_ptr + (row * num_cols) + col) = x;
```

Macro:

```
#define mat_ptr(r,c)  (*mat_ptr + ((r) *  
                                (num_cols) + (c)))
```

```
mat_ptr(row,col) = x;
```



Design Considerations

- Two important considerations
 - Choosing parameter-passing methods: efficiency
 - Whether one-way or two-way data transfer is needed
- Contemporary software-engineering principles dictate that access by subprogram code to data outside the subprogram should be minimized
- in-mode parameters should be used whenever no data are to be returned through parameters to the caller
- Out-mode parameters should be used when no data are transferred to the called subprogram but the subprogram must transmit data back to the caller
- inout-mode parameters should be used only when data must move in both directions between the caller and the called subprogram



Design Considerations

- There is a practical consideration that is in conflict with this principle
- Sometimes it is justifiable to pass access paths for one-way parameter transmission
 - **Eg:** When a large array is to be passed to a subprogram that does not modify it, a one-way method may be preferred
 - However, pass-by-value would require that the entire array be moved to a local storage area of the subprogram
 - This would be costly in both time and space
- Because of this, large arrays are often passed by reference
- C++ constant reference parameters offer another solution
- Another alternative approach would be to allow the user to choose between the **methods**
- The choice of a parameter-passing method for functions is related to another design issue: functional side effects

Static vs Dynamic Scoping

```

1   x <- 1
2   f <- function(a) x + a
3   g <- function() {
4     x <- 2
5     f(0)
6   }
7   g() # what does this return?
  
```

- Under *lexical scoping* (also known as *static scoping*), the scope of a variable is determined by the lexical (*i.e.*, textual) structure of a program
 - Under *dynamic scoping*, a variable is bound to the most recent value assigned to that variable
- In Static Scoping, the final result will be 1
 - In Dynamic Scoping, the final result will be 2



Static Scoping vs Dynamic Scoping

- Lexical scoping (sometimes known as *static scoping*) -> Sets the *scope* (range of functionality) of a variable so that it may only be *called* (referenced) from within the block of code in which it is defined
 - Scope is determined when the code is compiled
 - A variable declared in this fashion is sometimes called a *private* variable
- The opposite approach is known as *dynamic scoping* -> Creates variables that can be called from outside the block of code in which they are defined
 - A variable declared in this fashion is sometimes called a *public* variable



Private vs Public Variables

```
#include <iostream>
using namespace std;

// class definition
class Circle {
public: ←
    double radius;

    double compute_area()
    {
        return 3.14 * radius * radius;
    }
};

// main function
int main()
{
    Circle obj;

    // accessing public data member outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

O/P:

```
Radius is: 5.5
Area is: 94.985
```

```
#include <iostream>
using namespace std;

// class definition
class Circle {
private: ←
    double radius;

    double compute_area()
    {
        return 3.14 * radius * radius;
    }
};

// main function
int main()
{
    Circle obj;

    // accessing public data member outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

O/P: Error (Trying to access a variable and function which does not fall inside the scope)



Parameters that are Subprograms

- Subprogram names will be sent as parameters to other subprograms
- If only the transmission of the subprogram code was necessary, it could be done by passing a single pointer
- Two complications arise
 - Type checking the parameters of the activations of the subprogram that was passed as a parameter
 - In C and C++, functions cannot be passed as parameters, but pointers to functions can
 - Type of a pointer to a function includes the function's protocol
 - Because the protocol includes all parameter types, such parameters can be completely type checked



Miscellaneous

```
int main(void)
{
    char s1[80], s2[80];
    int (*p) (const char *, const char *);
    p = strcmp;
    printf("Enter two strings\n");
    gets(s1);
    gets(s2);
    check(s1, s2, p);
    return 0;
}
```

```
void check(char *a, char *b, int
(*cmp)(const char *, const
char *))
{
    printf("Testing for Equality\n");
    if(!(*cmp(a, b)))
        printf("Equal");
    else
        printf("Not Equal");
}
```



Parameters that are Subprograms

- Languages that allow nested subprograms -> What referencing environment for executing the passed subprogram should be used
 - Environment of the call statement that enacts the passed subprogram -> Shallow Binding
 - Environment of the definition of the passed subprogram -> Deep Binding
 - Environment of the call statement that passed the subprogram as an actual parameter -> Ad hoc Binding

Parameters that are Subprograms

- Consider the execution of sub2 when it is called in sub4
 - Shallow Binding -> Referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4 (*Output is 4*)
 - Deep Binding -> Referencing environment of sub2's execution is that of sub1, so the reference to x in sub2 is bound to the local x in sub1 (*Output is 1*)
 - Ad hoc Binding, the binding is to the local x in sub3 (*Output is 3*)

```

function sub1() {
  var x;
  function sub2() {
    alert(x); // Creates a dialog box with the value of x
  };
  function sub3() {
    var x;
    x = 3; ←
    sub4(sub2);
  };
  function sub4(subx) {
    var x;
    x = 4; ←
    subx();
  };
  x = 1; ←
  sub3();
}

```

- Environment of the call statement that enacts the passed subprogram -> Shallow Binding
- Environment of the definition of the passed subprogram -> Deep Binding
- Environment of the call statement that passed the subprogram as an actual parameter -> Ad hoc Binding



Parameters that are Subprograms

```
function sub1() {  
    var x;  
    function sub2() {  
        alert(x); // Creates a dialog box with the value of x  
    };  
    function sub3() {  
        var x;  
        x = 3;  
        sub4(sub2); //Assume this statement is  
        inserted at this location  
    };  
    function sub4(subx) {  
        var x;  
        x = 4;  
        subx();  
    };  
    x = 1;  
    sub3();  
};
```

Note: Here, both Deep Binding and Ad hoc Binding will output value 1. Because, sub2 is declared inside sub1. And, sub2 is passed as argument to sub4 by sub1 itself.

- Environment of the call statement that enacts the passed subprogram -> Shallow Binding
- Environment of the definition of the passed subprogram -> Deep Binding
- Environment of the call statement that passed the subprogram as an actual parameter -> Ad hoc Binding



Parameters that are Subprograms

- In some cases, the subprogram that declares a subprogram also passes that subprogram as a parameter
- In those cases, deep binding and ad hoc binding are the same
- Ad hoc binding has never been used because, one might surmise, the environment in which the procedure appears as a parameter has no natural connection to the passed subprogram
- Shallow binding is not appropriate for static-scoped languages with nested subprograms
- Eg: Suppose the procedure Sender passes the procedure Sent as a parameter to the procedure Receiver
- Problem is that Receiver may not be in the static environment of Sent, thereby making it very unnatural for Sent to have access to Receiver's variables
- On the other hand, it is perfectly normal in such a language for any subprogram, including one sent as a parameter, to have its referencing environment determined by the lexical position of its definition
- It is therefore more logical for these languages to use deep binding
- Some dynamic-scoped languages use shallow binding



Parameters that are Subprograms

```
function sub1() {  
    var x;  
    function sub2() {  
        alert(x); // Creates a dialog box with the value of x  
    };  
  
    function sub3() {  
        var x;  
        x = 3; ←  
        sub4(sub2);  
    };  
  
    function sub4(subx) {  
        var x;  
        x = 4; ←  
        subx();  
    };  
  
    x = 1; ←  
    sub3();  
};
```

Assume:

sub3 -> Sender
sub2 -> Sent
sub4 -> Receiver

- Shallow binding is not appropriate for static-scoped languages with nested subprograms
- Eg: Suppose the procedure Sender (sub3) passes the procedure Sent (sub2) as a parameter to the procedure Receiver (sub4)
- Problem is that Receiver (sub4) may not be in the static environment of Sent (sub2), thereby making it very unnatural for Sent (sub2) to have access to Receiver's (sub4's) variables
- On the other hand, it is perfectly normal in such a language for any subprogram, including one sent as a parameter, to have its referencing environment determined by the lexical position of its definition
- It is therefore more logical for these languages to use deep binding
- Some dynamic-scoped languages use shallow binding



Calling Subprograms Indirectly

- Occurs when the specific subprogram to be called is not known until run time
- Call to the subprogram is made through a pointer or reference to the subprogram, which has been set during execution before the call is made
- **Applications:** Event handling in graphical user interfaces; Callbacks
-> A subprogram is called and instructed to notify the caller when the called subprogram has completed its work
- C and C++ allow a program to define a pointer to a function, through which the function can be called
- In C++, pointers to functions are typed according to the return type and parameter types of the function, so that such a pointer can point only at functions with one particular protocol

```
float (*pfun)(float, int);
```

- In C and C++, a function name without following parentheses, like an array name without following brackets, is the address of the function (or array)



Calling Subprograms Indirectly

- Both of the following are legal ways of giving an initial value or assigning a value to a pointer to a function

```
int myfun2 (int, int); // A function declaration
int (*pfun2)(int, int) = myfun2; // Create a pointer and
                                // initialize
                                // it to point to myfun2
pfun2 = myfun2; // Assigning a function's address to a
                 // pointer
```

```
(*pfun2)(first, second);
pfun2(first, second);
```

- The first of these explicitly dereferences the pointer pfun2, which is legal, but unnecessary
- The function pointers of C and C++ can be sent as parameters and returned from functions, although functions cannot be used directly in either of those roles



Miscellaneous

```
int main(void)
{
    char s1[80], s2[80];
    int (*p) (const char *, const char *);
    p = strcmp;
    printf("Enter two strings\n");
    gets(s1);
    gets(s2);
    check(s1, s2, p);
    return 0;
}
```

```
void check(char *a, char *b, int
(*cmp)(const char *, const
char *))
{
    printf("Testing for Equality\n");
    if(!(*cmp(a, b)))
        printf("Equal");
    else
        printf("Not Equal");
}
```



Overloaded Subprograms

- Overloaded operator is one that has multiple meanings
- Meaning of a particular instance of an overloaded operator is determined by the types of its operands. **Eg:** $a * b$
- Overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment

```
int sum(int a, int b);  
int sum(int a, int b, int c);  
float sum(float a, float b);
```

- Every version of an overloaded subprogram must have a unique protocol
 - Must be different from the others in the number, order, or types of its parameters, and possibly in its return type if it is a function
- Meaning of a call to an overloaded subprogram is determined by the actual parameter list (and/or possibly the type of the returned value, in the case of a function)
- Not necessary that overloaded subprograms usually implement the same process



Overloaded Subprograms

- Because C++, Java, and C# allow mixed-mode expressions, the return type is irrelevant to disambiguation of overloaded functions (or methods)
- The context of the call does not allow the determination of the return type
- **Eg:** If a C++ program has two functions named fun and both take an int parameter but one returns an int and one returns a float, the program would not compile
 - Compiler could not determine which version of fun should be used

```
int sum(int a, int b);  
float sum(int a, int b); // Not valid
```

- Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls
- Call is ambiguous and will cause a compilation error

```
void fun(float b = 0.0);  
void fun();  
  
fun();
```



Design Issues for Functions

- Are side effects allowed? Eg: int Sum = a + fun(a)
- What types of values can be returned?
- How many values can be returned?

Functional Side Effects

- Because of the problems of side effects of functions that are called in expressions, parameters to functions should always be in-mode parameters (Ada)
 - Effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals
- Most other imperative languages, however, functions can have either pass-by-value or pass-by-reference parameters, thus allowing functions that cause side effects and aliasing
- Pure functional languages, such as Haskell, do not have variables, so their functions cannot have side effects



Design Issues for Functions

Types of Returned Values

- Most imperative programming languages restrict the types that can be returned by their functions
- C allows any type to be returned by its functions except arrays and functions
 - Both of these can be handled by pointer type return values
- C++ is like C but also allows user-defined types, or classes, to be returned from its functions
- In some programming languages, subprograms are first-class objects, which means that they can be passed as parameters, returned from functions, and assigned to variables
- Methods are first-class objects in some imperative languages, for example, Python, Ruby, and Lua
 - Same is true for the functions in most functional languages



Design Issues for Functions

Number of Returned Values

- In most languages, only a single value can be returned from a function
- However, that is not always the case
- Ruby allows the return of more than one value from a method
 - If a return statement in a Ruby method is not followed by an expression, nil is returned
 - If followed by one expression, the value of the expression is returned
 - If followed by more than one expression, an array of the values of all of the expressions is returned
- Lua also allows functions to return multiple values. Such values follow the return statement as a comma-separated list, as in the following:

return 3, sum, index

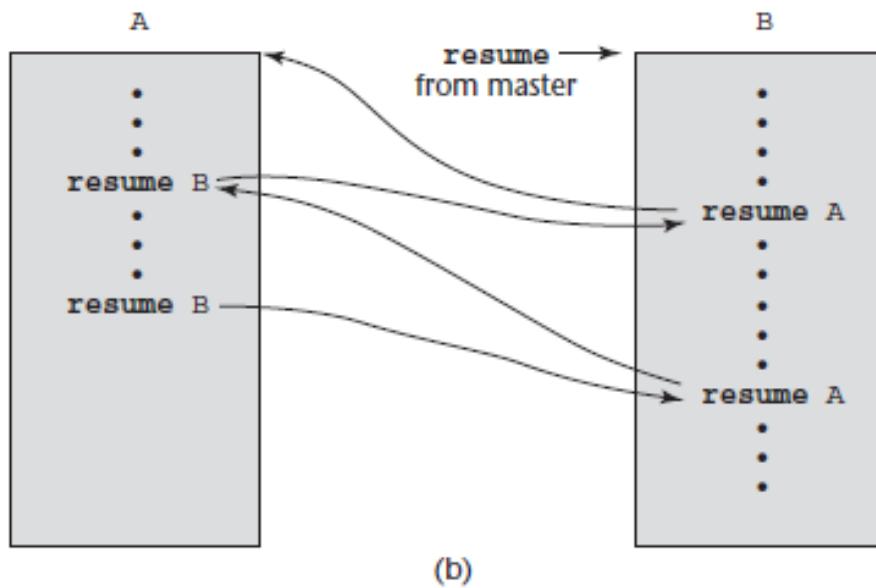
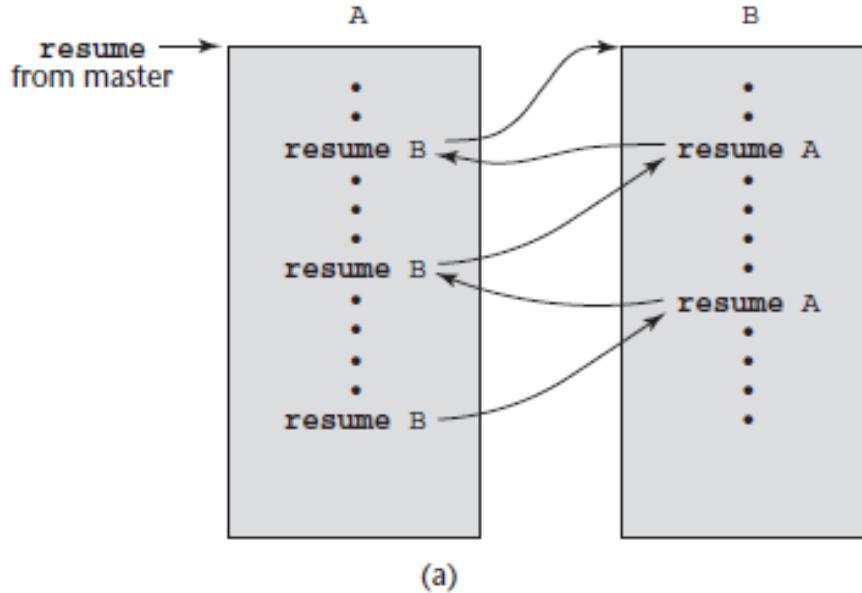


Design Issues for Functions

- The form of the statement that calls the function determines the number of values that are received by the caller
 - If the function is called as a procedure, that is, as a statement, all return values are ignored
 - If the function returned three values and all are to be kept by the caller, the function would be called as in the following example:

a, b, c = fun()

Coroutines





Coroutines

- Coroutine is a special kind of subprogram
- Rather than the master-slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are more equitable
- Coroutine control mechanism is often called the symmetric unit control model
- Coroutines can have multiple entry points, which are controlled by the coroutines themselves
- They also have the means to maintain their status between activations
- This means that coroutines must be history sensitive and thus have static local variables
- Secondary executions of a coroutine often begin at points other than its beginning
- Because of this, the invocation of a coroutine is called a resume rather than a call



Coroutines

```
sub co1 () {  
    ...  
    resume co2 () ;  
    ...  
    resume co3 () ;  
    ...  
}
```

- Only one coroutine is actually in execution at a given time
- Rather than executing to its end, a coroutine often partially executes and then transfers control to some other coroutine, and when restarted, a coroutine resumes execution just after the statement it used to transfer control elsewhere
- This sort of interleaved execution sequence is related to the way multiprogramming operating systems work
- Although there may be only one processor, all of the executing programs in such a system appear to run concurrently while sharing the processor
- In the case of coroutines, this is sometimes called quasi-concurrency



Coroutines

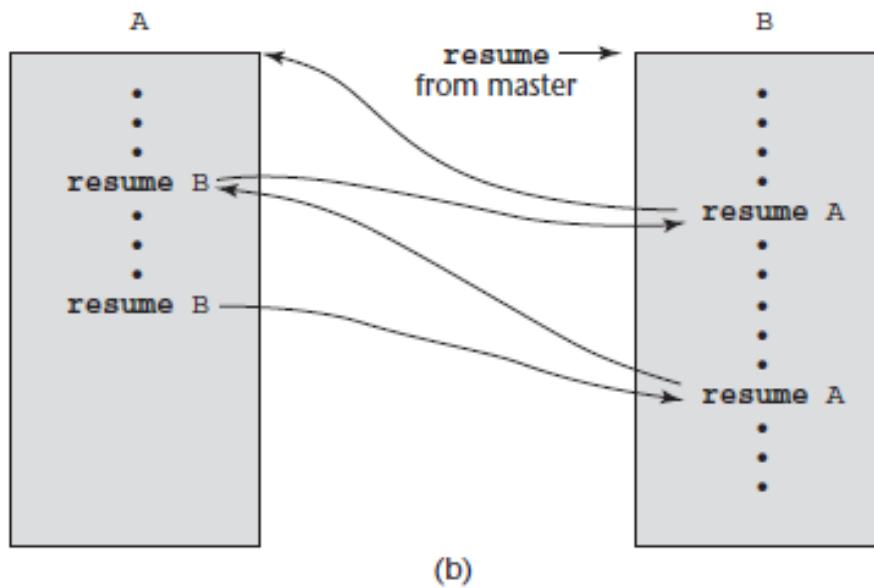
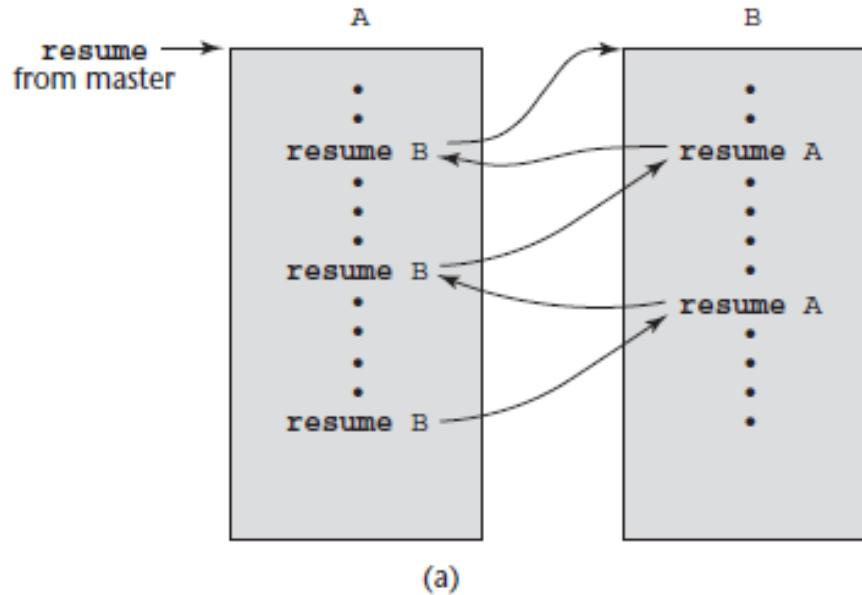
- Coroutines are created in an application by a program unit called the master unit, which is not a coroutine
- When created, coroutines execute their initialization code and then return control to that master unit
- When the entire family of coroutines is constructed, the master program resumes one of the coroutines, and the members of the family of coroutines then resume each other in some order until their work is completed, if in fact it can be completed
- If the execution of a coroutine reaches the end of its code section, control is transferred to the master unit that created it
- This is the mechanism for ending execution of the collection of coroutines, when that is desirable
- In some programs, the coroutines run whenever the computer is running



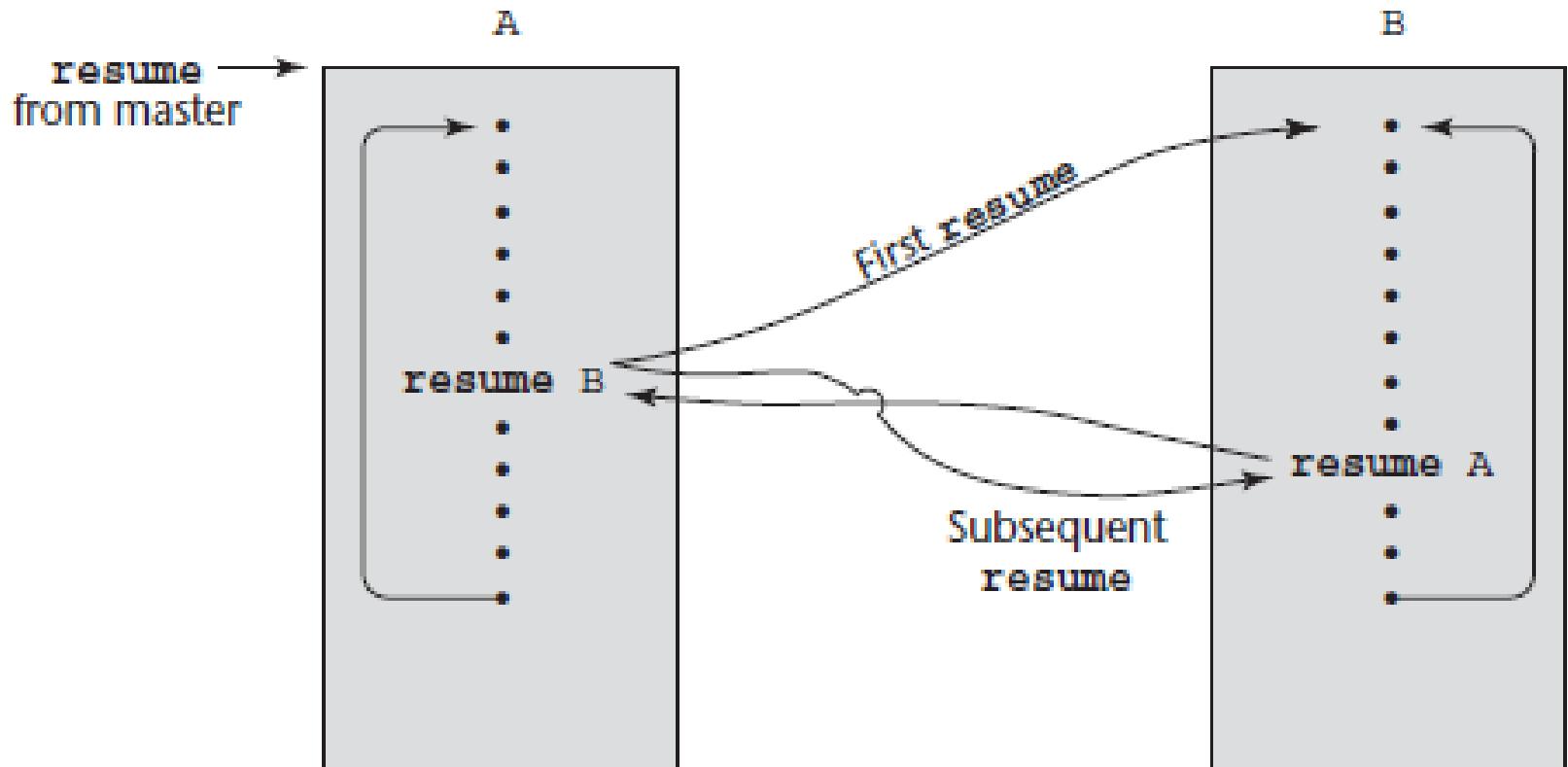
Coroutines

- One example of a problem that can be solved with this sort of collection of coroutines is a card game simulation
- Suppose the game has four players who all use the same strategy
- Such a game can be simulated by having a master program unit create a family of coroutines, each with a collection, or hand, of cards
- The master program could then start the simulation by resuming one of the player coroutines, which, after it had played its turn, could resume the next player coroutine, and so forth until the game ended
- Rather than have the patterns, a coroutine often has a loop containing a resume
- Among contemporary languages, only Lua fully supports coroutines

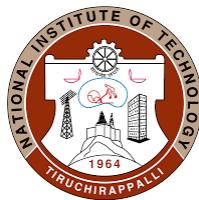
Coroutines



Coroutines



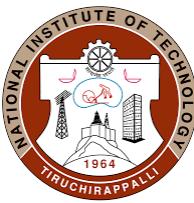
Rather than have the patterns, a coroutine often has a loop containing a resume



CSPC31: Principles of Programming Languages

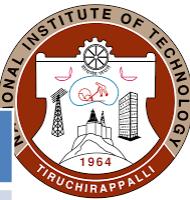
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 10 – Implementing Subprograms



Objectives

- Explore the implementation of subprograms
- How subprogram linkage works
- Nonnestable subprograms with static local variables
- Subprograms with stack-dynamic local variables
- Nested subprograms with stack-dynamic local variables and static scoping
- Static chain method of accessing nonlocals in static-scoped languages
- Techniques for implementing blocks
- Several methods of implementing nonlocal variable access in a dynamic-scoped language



Static vs Dynamic Scoping

```
1 x <- 1
2 f <- function(a) x + a
3 g <- function() {
4   x <- 2
5   f(0)
6 }
7 g() # what does this return?
```

- Under *lexical scoping* (also known as *static scoping*), the scope of a variable is determined by the lexical (*i.e.*, textual) structure of a program
 - Under *dynamic scoping*, a variable is bound to the most recent value assigned to that variable
- In Static Scoping, the final result will be 1
 - In Dynamic Scoping, the final result will be 2



Static Scoping vs Dynamic Scoping

- Lexical scoping (sometimes known as *static scoping*) is a convention used with many programming languages that sets the *scope* (range of functionality) of a variable so that it may only be *called* (referenced) from within the block of code in which it is defined. The scope is determined when the code is compiled. A variable declared in this fashion is sometimes called a *private* variable
- The opposite approach is known as *dynamic scoping*. Dynamic scoping creates variables that can be called from outside the block of code in which they are defined. A variable declared in this fashion is sometimes called a *public* variable



Introduction

- Subprogram call and return operations are together called subprogram linkage
- Call process must include the implementation of whatever parameter-passing method is used
- If local variables are not static, the call process must allocate storage for the locals declared in the called subprogram and bind those variables to that storage
- Must save the execution status of the calling program unit
- Execution status is everything needed to resume execution of the calling program unit -> Register Values, CPU Status Bits and the Environment Pointer (EP)
- EP is used to access parameters and local variables during the execution of a subprogram
- Calling process also must arrange to transfer control to the code of the subprogram and ensure that control can return to the proper place when the subprogram execution is completed

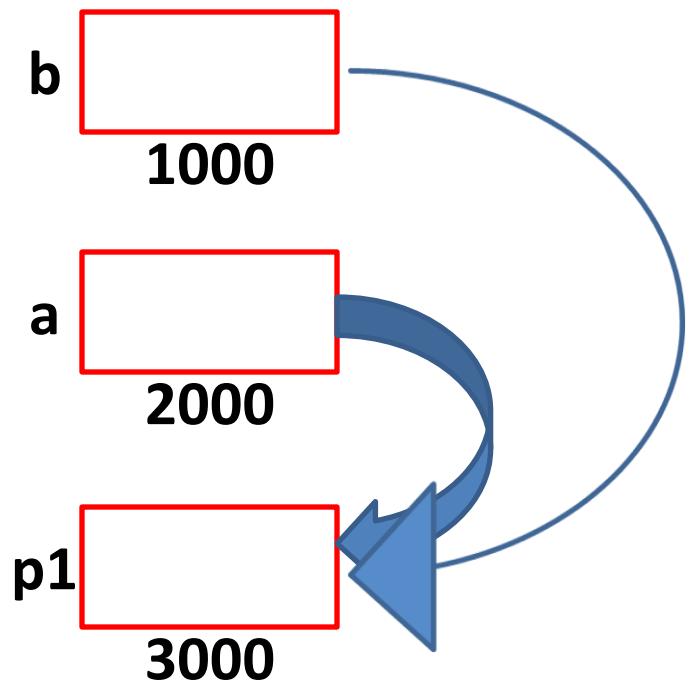


Introduction

- If the language supports nested subprograms, the call process must create some mechanism to provide access to nonlocal variables that are visible to the called subprogram
- Required actions of a subprogram return are less complicated than those of a call
- If the subprogram has parameters that are out mode or inout mode and are implemented by copy, the first action of the return process is to move the local values of the associated formal parameters to the actual parameters
- Next, it must deallocate the storage used for local variables and restore the execution status of the calling program unit
- Finally, control must be returned to the calling program unit

Pass-by-Result (Miscellaneous)

```
void sub(out int a, out int b)
{
    a = 17;
    b = 35;
}
int p1;
f.sub(out p1, out p1);
```





Implementing “Simple” Subprograms

- By “simple” we mean that subprograms cannot be nested and all local variables are static
- Semantics of a call to a “simple” subprogram requires the following actions
 - Save the execution status of the current program unit
 - Compute and pass the parameters
 - Pass the return address to the called
 - Transfer control to the called
- Semantics of a return from a simple subprogram requires the following actions
 - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters
 - If the subprogram is a function, the functional value is moved to a place accessible to the caller
 - Execution status of the caller is restored
 - Control is transferred back to the caller



Implementing “Simple” Subprograms

- The call and return actions require storage for the following
 - Status information about the caller
 - Parameters
 - Return address
 - Return value for functions
 - Temporaries used by the code of the subprograms
- These, along with the local variables and the subprogram code, form the complete collection of information a subprogram needs to execute and then return control to the caller
- Question now is the distribution of the call and return actions to the caller and the called



Implementing “Simple” Subprograms

- Last three actions of a call clearly must be done by the caller
- Saving the execution status of the caller could be done by either
- In the case of the return, the first, **second**, **third** and fourth actions must be done by the called
- Restoration of the execution status of the caller could be done by either the caller or the called
- In general, the linkage actions of the called can occur at two different times
 - At the beginning of execution -> Prologue of the subprogram linkage
 - At the end of execution -> Epilogue of the subprogram linkage
- In the case of a simple subprogram, all of the linkage actions of the callee occur at the end of its execution, so there is no need for a prologue



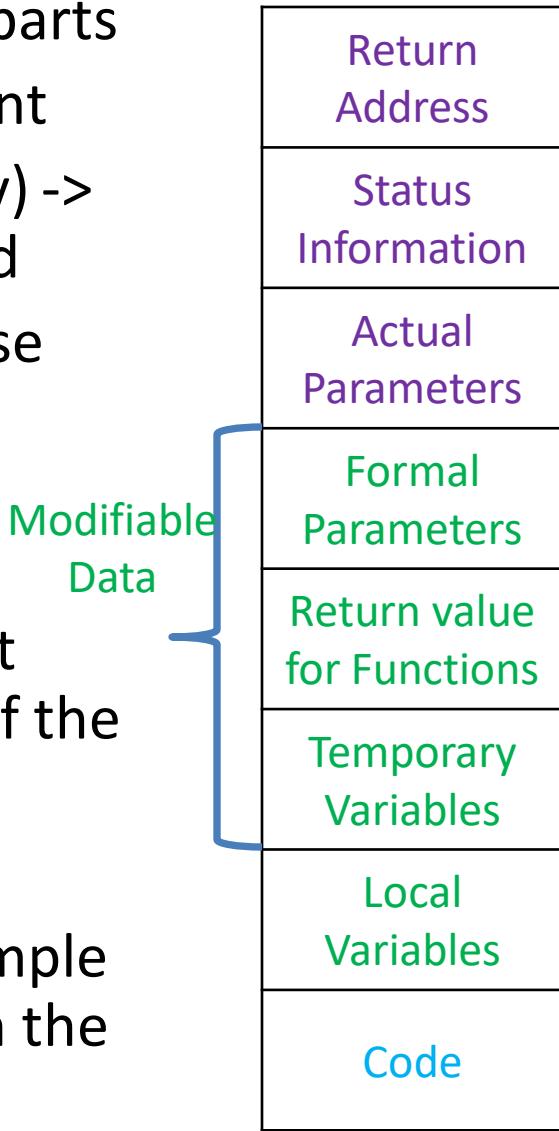
Miscellaneous

- Semantics of a call to a “simple” subprogram requires the following actions
 - Save the execution status of the current program unit
 - Compute and pass the parameters
 - Pass the return address to the called
 - Transfer control to the called
- Semantics of a return from a simple subprogram requires the following actions
 - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters
 - If the subprogram is a function, the functional value is moved to a place accessible to the caller
 - Execution status of the caller is restored
 - Control is transferred back to the caller



Implementing “Simple” Subprograms

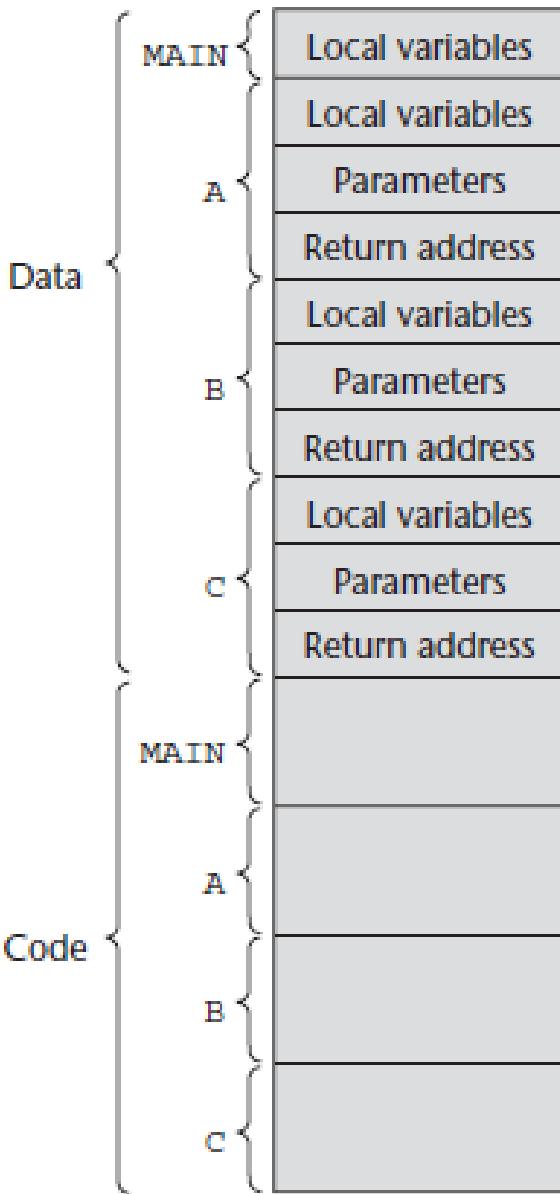
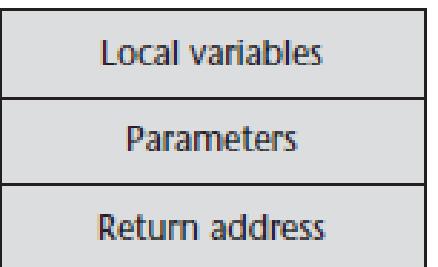
- A simple subprogram consists of two separate parts
 - Actual code of the subprogram -> Constant
 - Local variables and data (listed previously) -> Change when the subprogram is executed
- In the case of simple subprograms, both of these parts have fixed sizes
- Format or layout of the noncode part of a subprogram is called an activation record
 - Because the data it describes are relevant only during the activation, or execution of the subprogram
- Form of an activation record is static
- An activation record instance is a concrete example of an activation record -> A collection of data in the form of an activation record



Implementing “simple” Subprograms



- Because languages with simple subprograms do not support recursion, there can be only one active version of a given subprogram at a time
- Therefore, there can be only a single instance of the activation record for a subprogram
- Because an activation record instance for a “simple” subprogram has fixed size, it can be statically allocated
- In fact, it could be attached to the code part of the subprogram



Implementing “Simple” Subprograms



- Construction of the complete program is not done entirely by the compiler
- In fact, if the language allows independent compilation, the four program units have been compiled on different days, or even in different years
- At the time each unit is compiled, the machine code for it, along with a list of references to external subprograms, is written to a file
- Executable program is put together by the linker, which is part of the operating system (Sometimes linkers are called *loaders*, *linker/loaders* or *link editors*)
- When the linker is called for a main program, its first task is to find the files that contain the translated subprograms referenced in that program and load them into memory
- Then, the linker must set the target addresses of all calls to those subprograms in the main program to the entry addresses of those subprograms
- The same must be done for all calls to subprograms in the loaded subprograms and all calls to library subprograms
- In the previous example, the linker was called for MAIN
- The linker had to find the machine code programs for A, B, and C, along with their activation record instances, and load them into memory with the code for MAIN
- Then, it had to patch in the target addresses for all calls to A, B, C, and any library subprograms in A, B, C and MAIN



Implementing Subprograms with Stack-Dynamic Local Variables

- Examine the implementation of the subprogram linkage in languages in which locals are stack dynamic
- One of the most important advantages of stack-dynamic local variables is support for recursion

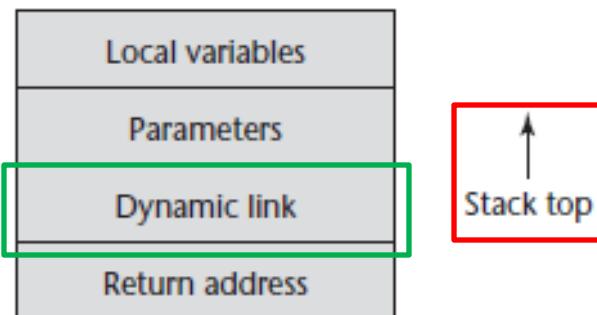
More Complex Activation Record

- Compiler must generate code to cause the implicit allocation and deallocation of local variables
- Recursion adds the possibility of multiple simultaneous activations of a subprogram, which means that there can be more than one instance (incomplete execution) of a subprogram at a given time, with at least one call from outside the subprogram and one or more recursive calls
 - Number of activations is limited only by the memory size of the machine
- Each activation requires its activation record instance



Implementing Subprograms with Stack-Dynamic Local Variables

- Format of an activation record for a given subprogram in most languages is known at compile time
- In many cases, the size is also known for activation records because all local data are of a fixed size
- That is not the case in some other languages, such as Ada, in which the size of a local array can depend on the value of an actual parameter
- In those cases, the format is static, but the size can be dynamic
- In languages with stack-dynamic local variables, activation record instances must be created dynamically





Miscellaneous

Stack ↓

(Return Address, Argument to Functions, Local Variables, Current CPU State)

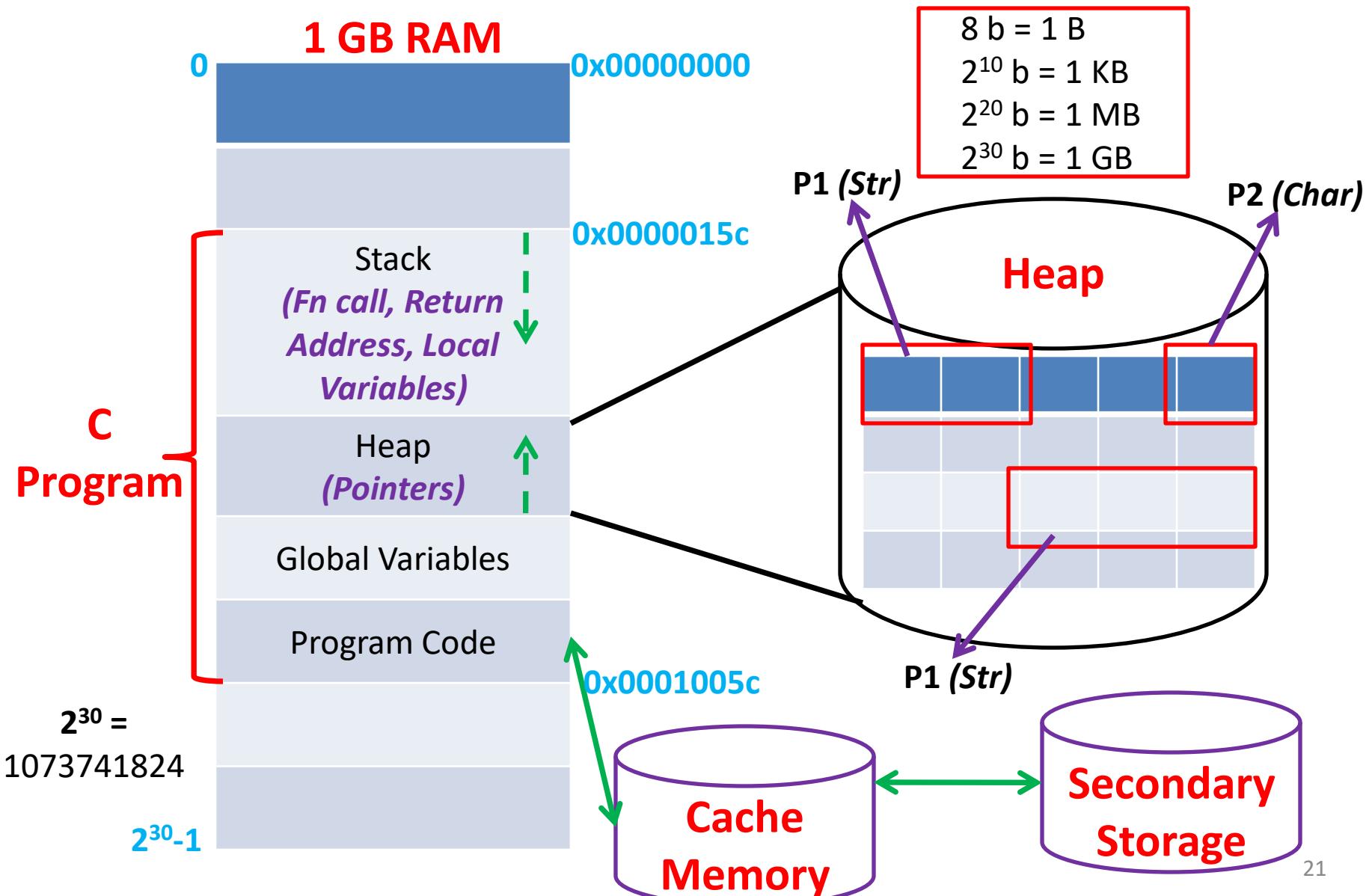
Heap ↑

(Dynamic Memory Allocation – malloc, realloc, free)

Global Variables

Program Code

Miscellaneous





Implementing Subprograms with Stack-Dynamic Local Variables

- Dynamic link is a pointer to the base of the activation record instance of the caller
 - In static-scoped languages, this link is used to provide traceback information when a run-time error occurs
 - In dynamic-scoped languages, the dynamic link is used to access nonlocal variables
- Actual parameters in the activation record are the values or addresses provided by the caller
- Local scalar variables are bound to storage within an activation record instance
- Local variables that are structures are sometimes allocated elsewhere, and only their descriptors and a pointer to that storage are part of the activation record
- Local variables are allocated and possibly initialized in the called subprogram, so they appear last

Scalar variable is a variable that holds only one value at a time



Implementing Subprograms with Stack-Dynamic Local Variables

- Format of the activation record is fixed at compile time, although its size may depend on the call in some languages
- Because the call and return semantics specify that the subprogram last called is the first to complete, it is reasonable to create instances of these activation records on a stack
- This stack is part of the runtime system and therefore is called the run-time stack, although we will usually just refer to it as the stack
- Every subprogram activation, whether recursive or nonrecursive, creates a new instance of an activation record on the stack
- This provides the required separate copies of the parameters, local variables and return address

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

Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Return address	

Implementing Subprograms with Stack-Dynamic Local Variables



- Initially, the EP points at the base, or first address of the activation record instance of the main program
- Therefore, the run-time system must ensure that it always points at the base of the activation record instance of the currently executing program unit
- When a subprogram is called, the current EP is saved in the new activation record instance as the dynamic link
- The EP is then set to point at the base of the new activation record instance
- Upon return from the subprogram, the stack top is set to the value of the **current EP minus one** and the EP is set to the dynamic link from the activation record instance of the subprogram that has completed its execution
- Resetting the stack top effectively removes the top activation record instance

Miscellaneous

```

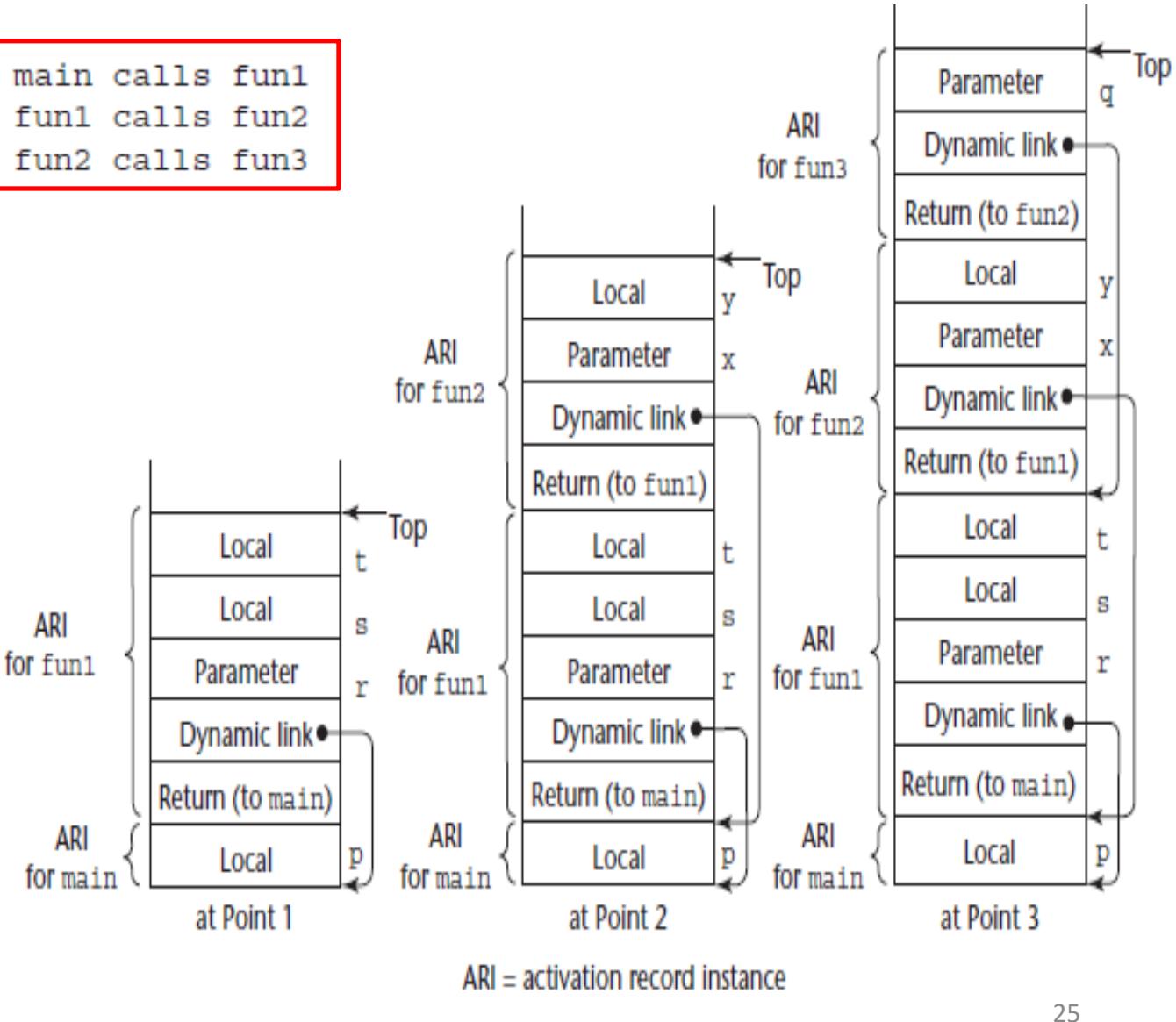
void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}

void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}

void fun3(int q) {
    ...
}

void main() {
    float p;
    ...
    fun1(p);
    ...
}
  
```

main calls fun1
 fun1 calls fun2
 fun2 calls fun3





Implementing Subprograms with Stack-Dynamic Local Variables

- EP is used as the base of the offset addressing of the data contents of the activation record instance -> Parameters and Local variables
- Note that the EP currently being used is not stored in the run-time stack
- Only saved versions are stored in the activation record instances as the dynamic links
- Using the activation record form given in this section, the new actions are as follows
- Caller actions are as follows
 - Create an activation record instance
 - Save the execution status of the current program unit
 - Compute and pass the parameters
 - Pass the return address to the called
 - Transfer control to the called



Implementing Subprograms with Stack-Dynamic Local Variables

- The prologue actions of the called are as follows
 - Save the old EP in the stack as the dynamic link and create the new value
 - Allocate local variables
- The epilogue actions of the called are as follows
 - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
 - If the subprogram is a function, the functional value is moved to a place accessible to the caller
 - Restore the stack pointer by setting it to the value of the **current EP minus one** and set the EP to the old dynamic link
 - Restore the execution status of the caller
- A subprogram is active from the time it is called until the time that execution is completed
- At the time it becomes inactive, its local scope ceases to exist and its referencing environment is no longer meaningful
- Therefore, at that time, its activation record instance can be destroyed



Implementing Subprograms with Stack-Dynamic Local Variables

- Parameters are not always transferred in the stack
 - In many compilers for RISC (Reduced Instruction Set Computer) machines, parameters are passed in registers
 - This is because RISC machines normally have many more registers than CISC (Complex Instruction Set Computer) machines
- Assume that parameters are passed in the stack
- It is straightforward to modify this approach for parameters being passed in registers

Implementing Subprograms with Stack-Dynamic Local Variables



An Example Without Recursion

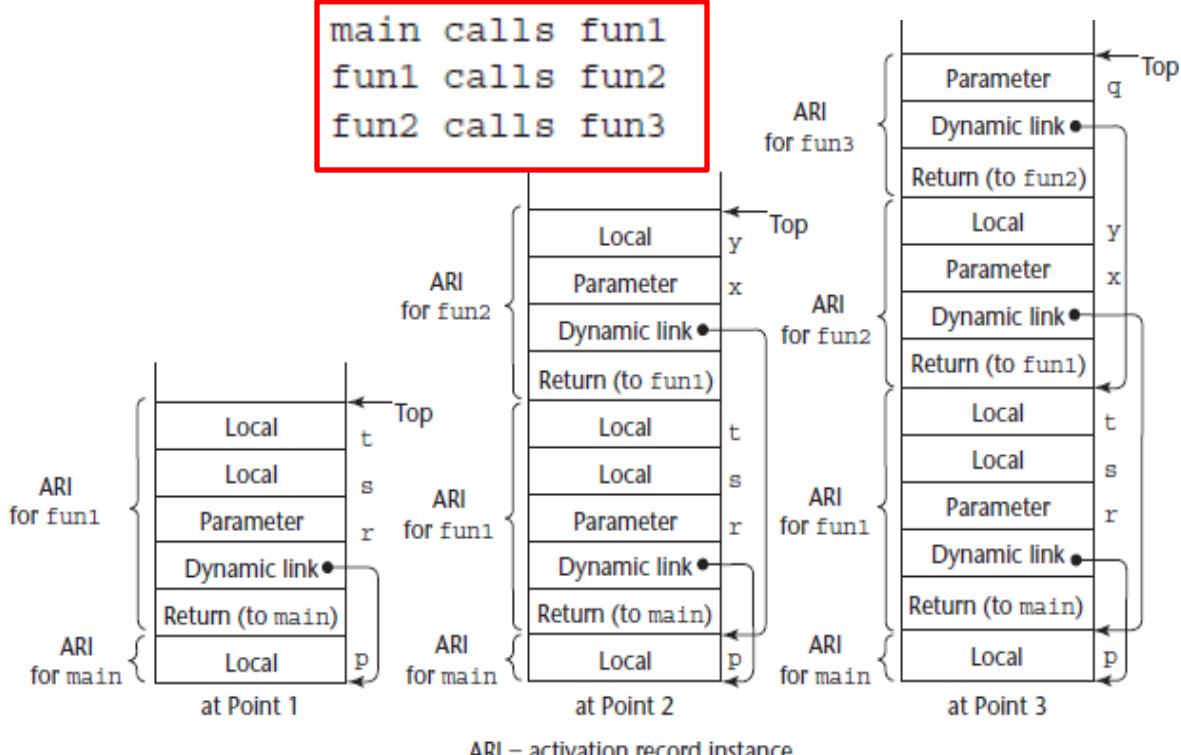
```

void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}

void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}

void fun3(int q) {
    ...
}

void main() {
    float p;
    ...
    fun1(p);
    ...
}
  
```



- Some implementations do not actually use an activation record instance on the stack for main functions
- Assume that the stack grows from lower addresses to higher addresses, although in a particular implementation, the stack may grow in the opposite direction

Implementing Subprograms with Stack-Dynamic Local Variables



- Collection of dynamic links present in the stack at a given time is called the dynamic chain or call chain
- It represents the dynamic history of how execution got to its current position, which is always in the subprogram code whose activation record instance is on top of the stack
- References to local variables can be represented in the code as offsets from the beginning of the activation record of the local scope, whose address is stored in the EP
- Such an offset is called a local_offset
- Local_offset of a variable in an activation record can be determined at compile time, using the order, types, and sizes of variables declared in the subprogram associated with the activation record



Implementing Subprograms with Stack-Dynamic Local Variables

- We assume that all variables take one position in the activation record
 - First local variable declared in a subprogram would be allocated in the activation record two positions plus the number of parameters from the bottom (the first two positions are for the return address and the dynamic link)
 - The second local variable declared would be one position nearer the stack top and so forth
 - **Eg:** In fun1, the local_offset of s is 3; for t it is 4
 - Likewise, in fun2, the local_offset of y is 3
- To get the address of any local variable, the local_offset of the variable is added to the EP

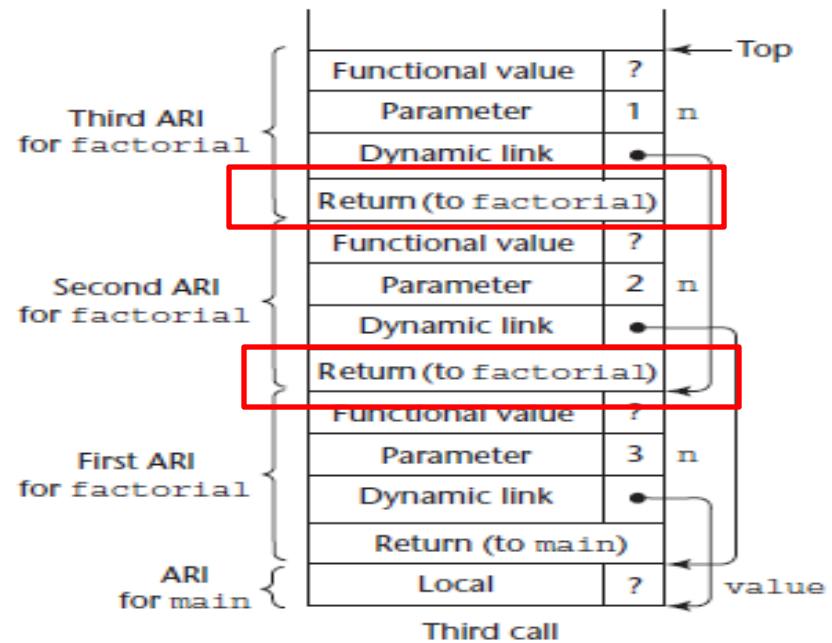
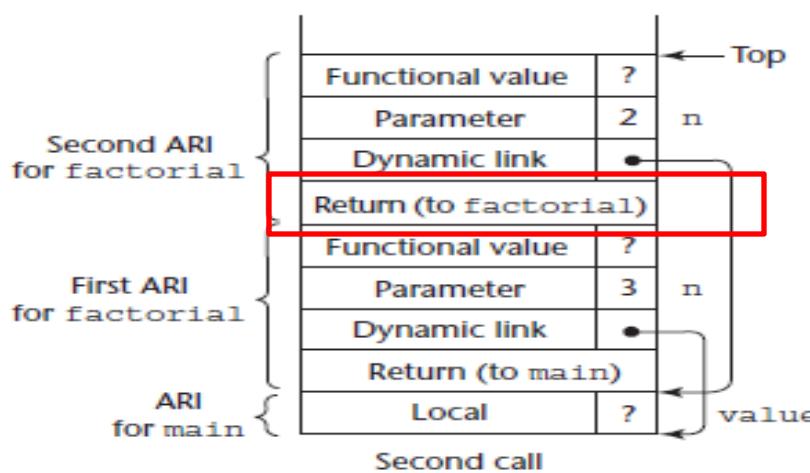
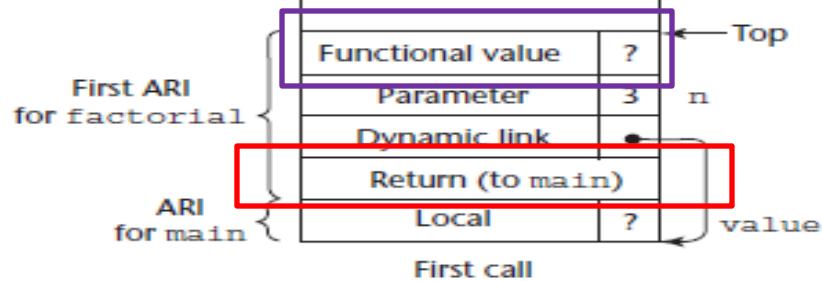
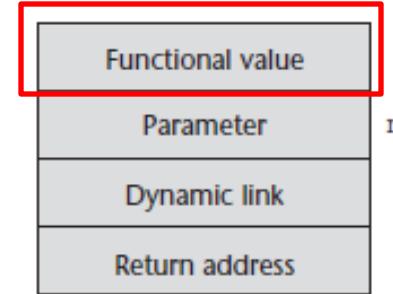
Implementing Subprograms with

Stack-Dynamic Local Variables

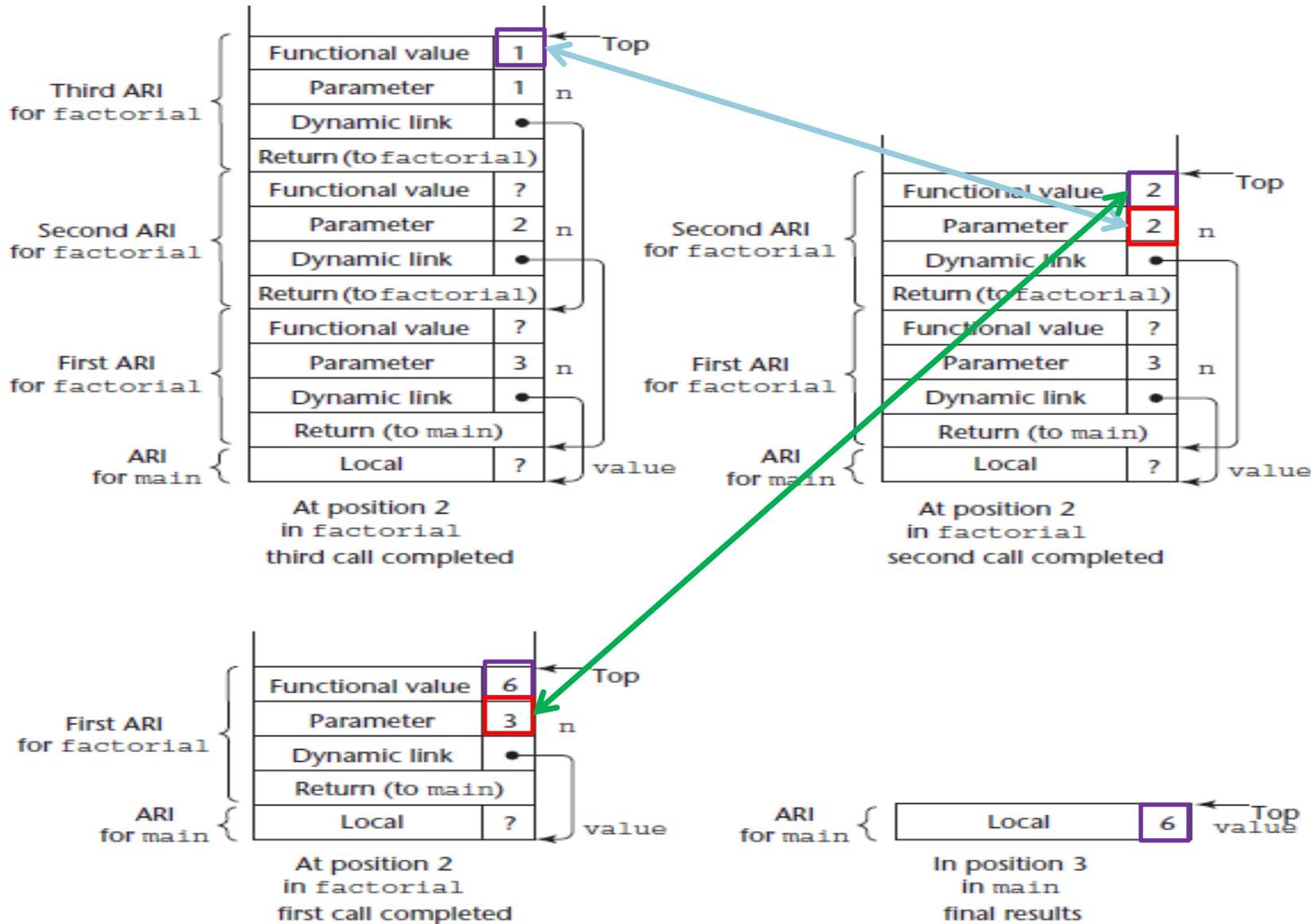


With Recursion

```
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
}
```



Implementing Subprograms with Stack-Dynamic Local Variables





Nested Subprograms

- Some of the non-C-based static-scoped programming languages use stack-dynamic local variables and allow subprograms to be nested
- Among these are Fortran 95+, Ada, Python, JavaScript, Ruby and Lua, as well as the functional languages

The Basics

- A reference to a nonlocal variable in a static-scoped language with nested subprograms requires a two-step access process
- All nonstatic variables that can be nonlocally accessed are in existing activation record instances and therefore are somewhere in the stack
- First step of the access process is to find the instance of the activation record in the stack in which the variable was allocated
- Second part is to use the local_offset of the variable (within the activation record instance) to access it



Miscellaneous

```
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 -- of Main_2
                    ...
                    Bigsub;
                    ...
                end; -- of Main_2
```



Nested Subprograms

- Finding the correct activation record instance is the more interesting and more difficult of the two steps
- First, note that in a given subprogram, only variables that are declared in static ancestor scopes are visible and can be accessed
- Also, activation record instances of all of the static ancestors are always on the stack when variables in them are referenced by a nested subprogram
- This is guaranteed by the static semantic rules of the static-scoped languages:
 - A subprogram is callable only when all of its static ancestor subprograms are active
 - If a particular static ancestor were not active, its local variables would not be bound to storage, so it would be nonsense to allow access to them
- The semantics of nonlocal references dictates that the correct declaration is the first one found when looking through the enclosing scopes, most closely nested first
- So, to support nonlocal references, it must be possible to find all of the instances of activation records in the stack that correspond to those static ancestors



Nested Subprograms

Static Chains

- Most common way to implement static scoping in languages that allow nested subprograms is static chaining
- A new pointer, called a static link, is added to the activation record
- Static link, which is sometimes called a *static scope pointer*, points to the bottom of the activation record instance of an activation of the static parent
- It is used for accesses to nonlocal variables
- The static link appears in the activation record below the parameters
- The addition of the static link to the activation record requires that local offsets differ from when the static link is not included
- Instead of having two activation record elements before the parameters, there are now three -> Return address, Static link and dynamic link
- A static chain is a chain of static links that connect certain activation record instances in the stack



Nested Subprograms

- Finding the correct activation record instance of a nonlocal variable using static links is relatively straightforward
- Because the nesting of scopes is known at compile time, the compiler can determine not only that a reference is nonlocal but also the length of the static chain that must be followed to reach the activation record instance that contains the nonlocal object
- Let `static_depth` be an integer associated with a static scope that indicates how deeply it is nested in the outermost scope
- A program unit that is not nested inside any other unit has a `static_depth` of 0
- If subprogram A is defined in a nonnested program unit, its `static_depth` is 1
- If subprogram A contains the definition of a nested subprogram B, then B's `static_depth` is 2

```
# Global scope
...
def f1():
    def f2():
        def f3():
            ...
            # end of f3
            ...
            # end of f2
            ...
            # end of f1
```



Nested Subprograms

- The length of the static chain needed to reach the correct activation record instance for a nonlocal reference to a variable X is exactly the difference between the `static_depth` of the subprogram containing the reference to X and the `static_depth` of the subprogram containing the declaration for X
 - This difference is called the `nesting_depth` or `chain_offset`, of the reference
- The actual reference can be represented by an ordered pair of integers (`chain_offset`, `local_offset`), where `chain_offset` is the number of links to the correct activation record instance
- The `static_depths` of the **global scope**, **f1**, **f2**, and **f3** are **0**, **1**, **2**, and **3**
 - If procedure f3 references a variable declared in f1, the `chain_offset` of that reference would be 2 (`static_depth` of f3 minus the `static_depth` of f1)
 - If procedure f3 references a variable declared in f2, the `chain_offset` of that reference would be 1
- References to locals can be handled using the same mechanism, with a `chain_offset` of 0, but instead of using the static pointer to the activation record instance of the subprogram where the variable was declared as the base address, the EP is used

Nested Subprograms

```

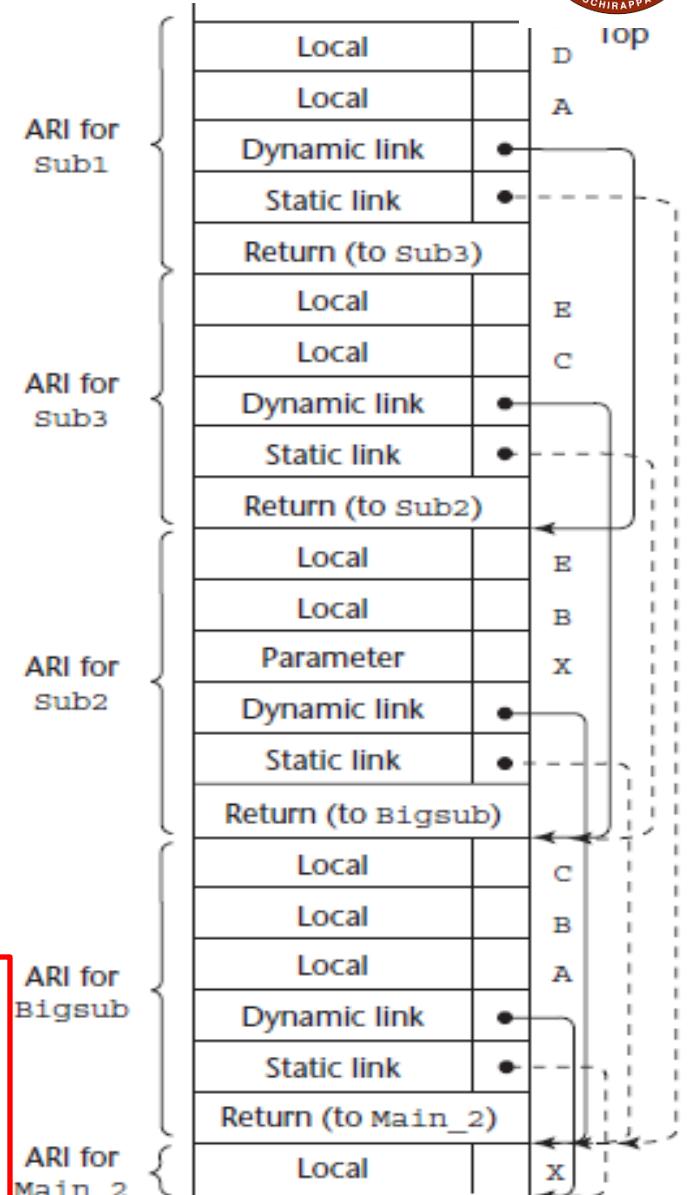
procedure Main_2 is 0
  X : Integer;
  procedure Bigsub is 1
    A, B, C : Integer;
    procedure Sub1 is 2
      A, D : Integer;
      begin -- of Sub1
        A := B + C; ← 1
        ...
      end; -- of Sub1
    procedure Sub2(X : Integer) is 3
      B, E : Integer;
      procedure Sub3 is
        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 -- of Main_2
    ...
    Bigsub;
    ...
end; -- of Main_2

```

Read Page No.
from 457 to 460

Main_2 calls Bigsub
Bigsub calls Sub2
Sub2 calls Sub3
Sub3 calls Sub1

References to the variable A at points 1, 2, and 3 would be represented by the following points:
(0, 3) (local)
(2, 3) (two levels away)
(1, 3) (one level away)





Blocks

- C-based languages, provide for user-specified local scopes for variables called blocks

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

- Lifetime of the variable temp in the preceding block begins when control enters the block and ends when control exits the block
- Advantage of using such a local is that it cannot interfere with any other variable with the same name that is declared elsewhere in the program, or more specifically, in the referencing environment of the block
- Blocks can be implemented by using the static-chain process



Blocks

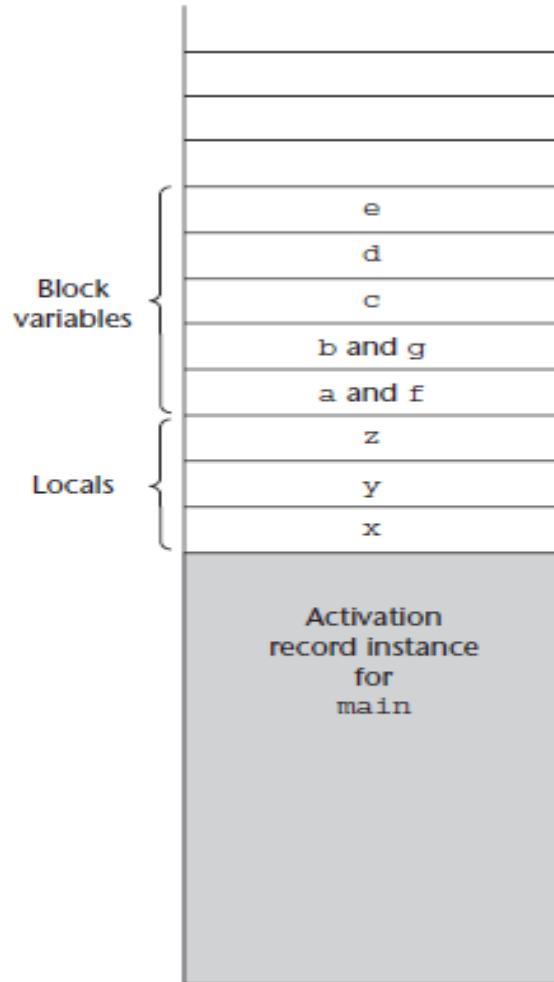
- Blocks are treated as parameterless subprograms that are always called from the same place in the program
 - Therefore, every block has an activation record
 - An instance of its activation record is created every time the block is executed
- Maximum amount of storage required for block variables at any time during the execution of a program can be statically determined, because blocks are entered and exited in strictly textual order
- This amount of space can be allocated after the local variables in the activation record
- Offsets for all block variables can be statically computed, so block variables can be addressed exactly as if they were local variables

Blocks

- Note that f and g occupy the same memory locations as a and b, because a and b are popped off the stack when their block is exited (before f and g are allocated)

```

void main() {
    int x, y, z;
    while ( ... ) {
        int a, b, c;
        ...
        while ( ... ) {
            int d, e;
            ...
        }
    }
    while ( ... ) {
        int f, g;
        ...
    }
    ...
}
  
```





Static vs Dynamic Scoping

```
1 x <- 1
2 f <- function(a) x + a
3 g <- function() {
4   x <- 2
5   f(0)
6 }
7 g() # what does this return?
```

- Under *lexical scoping* (also known as *static scoping*), the scope of a variable is determined by the lexical (*i.e.*, textual) structure of a program
 - Under *dynamic scoping*, a variable is bound to the most recent value assigned to that variable
-
- In Static Scoping, the final result will be 1
 - In Dynamic Scoping, the final result will be 2



Implementing Dynamic Scoping

- Two distinct ways in which local variables and nonlocal references to them can be implemented in a dynamic-scoped language -> Deep access; Shallow access
- Deep access and shallow access are not concepts related to deep and shallow binding
- An important difference between binding and access is that deep and shallow bindings result in different semantics
- Deep and shallow accesses do not

Deep Access

- If local variables are stack dynamic and are part of the activation records in a dynamic-scoped language, references to nonlocal variables can be resolved by searching through the activation record instances of the other subprograms that are currently active, beginning with the one most recently activated
- This concept is similar to that of accessing nonlocal variables in a static-scoped language with nested subprograms, except that the dynamic—rather than the static—chain is followed



Implementing Dynamic Scoping

- Dynamic chain, links together all subprogram activation record instances in the reverse of the order in which they were activated
- Therefore, the dynamic chain is exactly what is needed to reference nonlocal variables in a dynamic-scoped language
- This method is called deep access, because access may require searches deep into the stack
- Consider the references to the variables x, u, and v in function sub3
 - Reference to x is found in the activation record instance for sub3
 - Reference to u is found by searching *all of the activation record instances on* the stack, because the only existing variable with that name is in main
 - This search involves following four dynamic links and examining **9
10** variable names
 - Reference to v is found in the most recent (nearest on the dynamic chain) activation record instance for the subprogram sub1

Implementing Dynamic Scoping

```

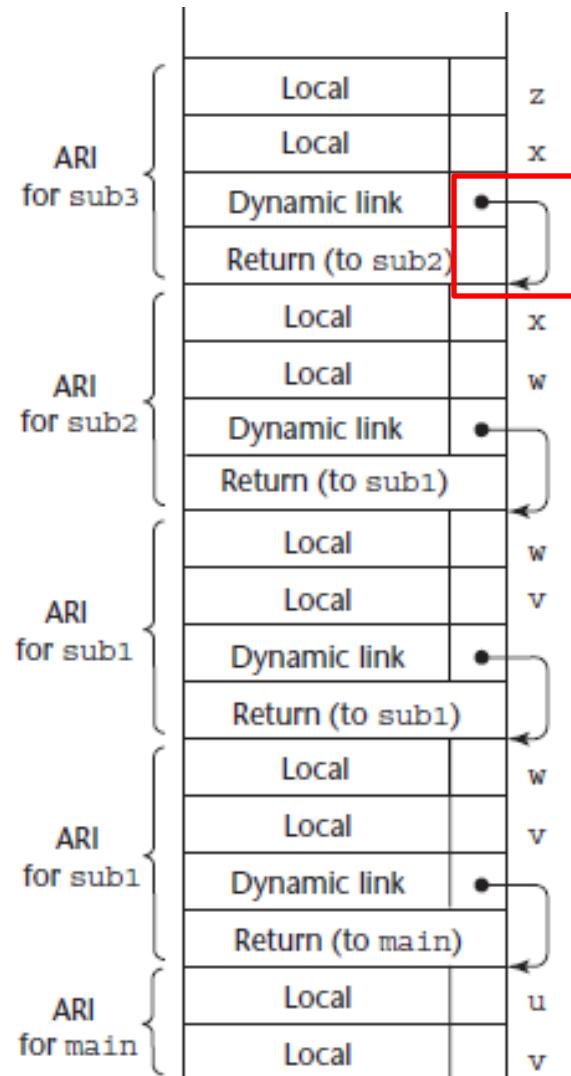
void sub3 () {
    int x, z;
    x = u + v; ←
    ...
}

void sub2 () {
    int w, x;
    ...
}

void sub1 () {
    int v, w;
    ...
}

void main () {
    int v, u;
    ...
}
    
```

main calls sub1
 sub1 calls sub1
 sub1 calls sub2
 sub2 calls sub3





Implementing Dynamic Scoping

- Two important differences between the deep-access method for nonlocal access in a dynamic-scoped language and the static-chain method for static-scoped languages
 - First, in a dynamic-scoped language, there is no way to determine at compile time the length of the chain that must be searched
 - ✓ Every activation record instance in the chain must be searched until the first instance of the variable is found
 - ✓ This is one reason why dynamic-scoped languages typically have slower execution speeds than static-scoped languages
 - Second, activation records must store the names of variables for the search process, whereas in static-scoped language implementations only the values are required (Names are not required for static scoping, because all variables are represented by the **chain_offset**, **local_offset** pairs)



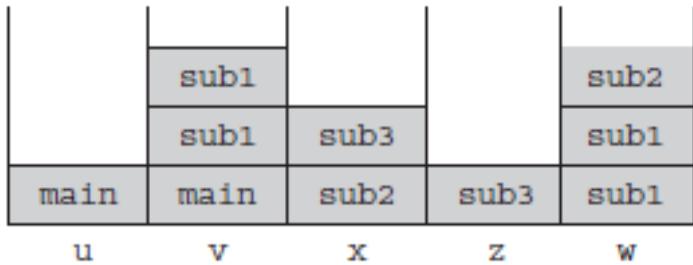
Implementing Dynamic Scoping

- Shallow access is an alternative implementation method
- Variables declared in subprograms are not stored in the activation records of those subprograms
- Because with dynamic scoping there is at most one visible version of a variable of any specific name at a given time, a very different approach can be taken

Method 1

- Have a separate stack for each variable name in a complete program
- Every time a new variable with a particular name is created by a declaration at the beginning of a subprogram that has been called, the variable is given a cell at the top of the stack for its name
- Every reference to the name is to the variable on top of the stack associated with that name, because the top one is the most recently created
- When a subprogram terminates, the lifetimes of its local variables end, and the stacks for those variable names are popped
- This method allows fast references to variables, but maintaining the stacks at the entrances and exits of subprograms is costly

Implementing Dynamic Scoping



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

Central Table

Variable	Value	Active
u	- 5	1
v	15	1
x	20	1
z	-25	1
w	12	1

Method 2

- Use a central table that has a location for each different variable name in a program
- Along with each entry, a bit called active is maintained that indicates whether the name has a current binding or variable association
- Any access to any variable can then be to an offset into the central table
- Offset is static, so the access can be fast



Miscellaneous

```
void sub3() {  
    int x, z;  
    x = u + v;  
    ...  
}
```

```
void sub2() {  
    int w, x;  
    ...  
}
```

```
void sub1() {  
    int v, w;  
    ...  
}
```

```
void main() {  
    int v, u;  
    ...  
}
```

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3



Implementing Dynamic Scoping

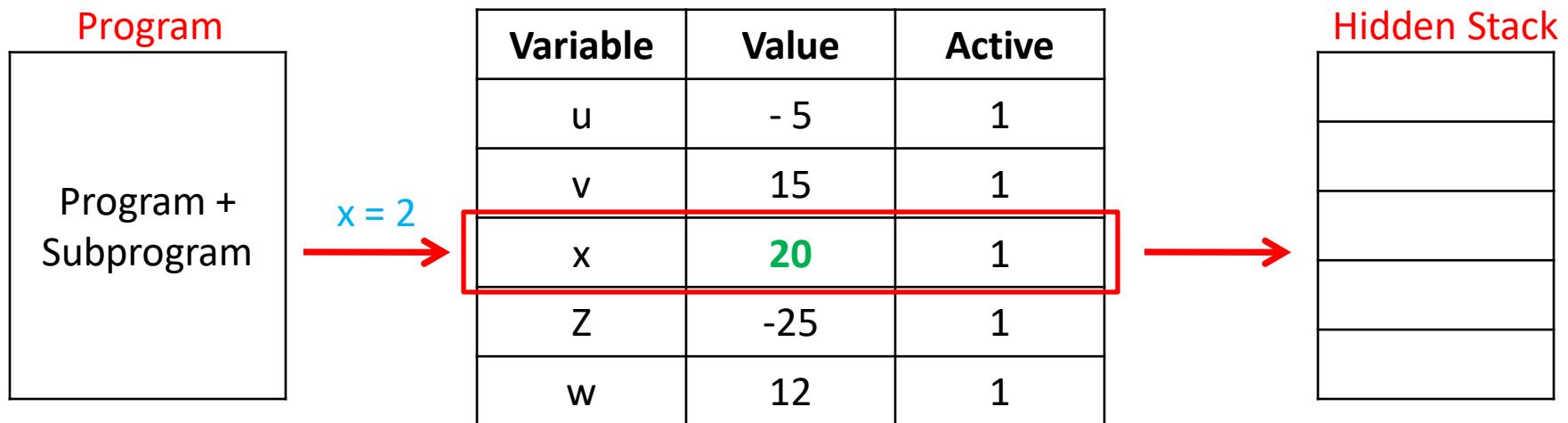
- Maintenance of a central table is straightforward
- A subprogram call requires that all of its local variables be logically placed in the central table
- If the position of the new variable in the central table is already active—that is, if it contains a variable whose lifetime has not yet ended (which is indicated by the active bit)—that value must be saved somewhere during the lifetime of the new variable
- Whenever a variable begins its lifetime, the active bit in its central table position must be set
- There have been several variations in the design of the central table and in the way values are stored when they are temporarily replace

Method 2 - Technique 1

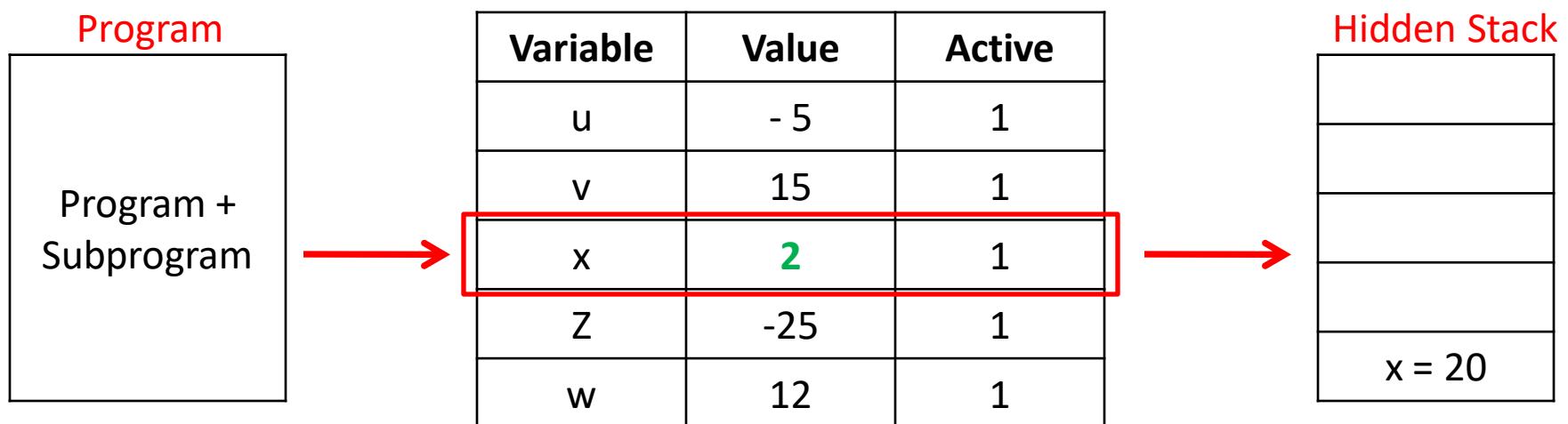
- One variation is to have a “hidden” stack on which all saved objects are stored
- Because subprogram calls and returns, and thus the lifetimes of local variables, are nested, this works well

Implement Dynamic Scoping

Central Table



Central Table





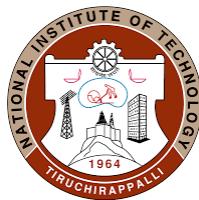
Implementing Dynamic Scoping

Method 2 - Technique 2

- A central table of single cells is used, storing only the current version of each variable with a unique name
- Replaced variables are stored in the activation record of the subprogram that created the replacement variable
- This is a stack mechanism, but it uses the stack that already exists, so the new overhead is minimal

Discussion

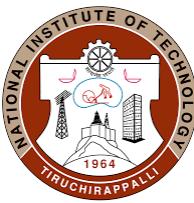
- Choice between shallow and deep access to nonlocal variables depends on the relative frequencies of subprogram calls and nonlocal references
 - Deep-access method provides fast subprogram linkage, but references to nonlocals, especially references to distant nonlocals (in terms of the call chain), are costly
 - Shallow-access method provides much faster references to nonlocals, especially distant nonlocals, but is more costly in terms of subprogram linkage



CSPC31: Principles of Programming Languages

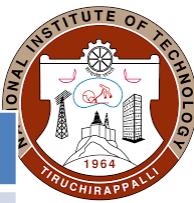
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 12 – Support for Object-Oriented Programming



Objectives

- Introduction to object-oriented programming
- Discussion of the primary design issues for inheritance and dynamic binding
- Support for object-oriented programming in C++
- Overview of the implementation of dynamic bindings of method calls to methods in object-oriented languages



Micsellaneous

Object-Oriented Programming Concepts

- Encapsulation
- Abstract Data Types
- Inheritance
- Polymorphism (Dynamic Binding)



Miscellaneous

```
function add(int c, int d)
{
    return (c + d);
}

void main()
{
    int a = 5, b = 10;
    printf("%d", add(a, b));
}
```

O/P: 15

```
class Add
{
public:
    int num1, num2;
    int sum(int n1, int n2)
    {
        return n1+n2;
    }
int main()
{
    //Creating object of class
    Add obj;
    obj.num1 = 5;
    obj.num2 = 10;
    cout << sum(num1, num2);
}
```

O/P: 15

```
class Add
{
protected:
    int num1, num2;
    int sum(int n1, int n2)
    {
        return n1 + n2;
    }
int main()
{
    //Creating object of class
    Add obj;
    obj.num1 = 5;
    obj.num2 = 10;
    cout << sum(num1, num2);
}
```

O/P: Error



Miscellaneous

```
class construct
{
public:
    int a, b;

    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called
    automatically when the object is
    created
    construct c;
    cout << "a: " << c.a << endl
        << "b: " << c.b;
    return 1;
}
```

O/P: a: 10 b: 20

```
class Point
{
private:
    int x, y;

public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX()
    {
        return x;
    }

    int getY()
    {
        return y;
    }
};
```

```
int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values
    assigned by Constructor
    cout << "p1.x = " <<
    p1.getX() << ", p1.y = " <<
    p1.getY();

    return 0;
}
```

O/P: p1.x = 10, p1.y = 15

Types

- Default Constructor
- Parameterized Constructor



Miscellaneous

```
class HelloWorld
{
public:
    HelloWorld()
    {
        cout<<"Constructor is called"\;
    }
    ~HelloWorld()
    {
        cout<<"Destructor is called"\;
    }
    void display()
    {
        cout<<"Hello World!"\;
    }
};
```

```
int main()
{
    HelloWorld obj;
    obj.display();
    return 0;
}
```

Destructor

A destructor is automatically called when:

- 1) The program finished execution
- 2) When a scope (the { } parenthesis) containing local variable ends
- 3) When you call the delete operator

O/P: Constructor is called
Hello World!
Destructor is called



Miscellaneous

Access Specifiers

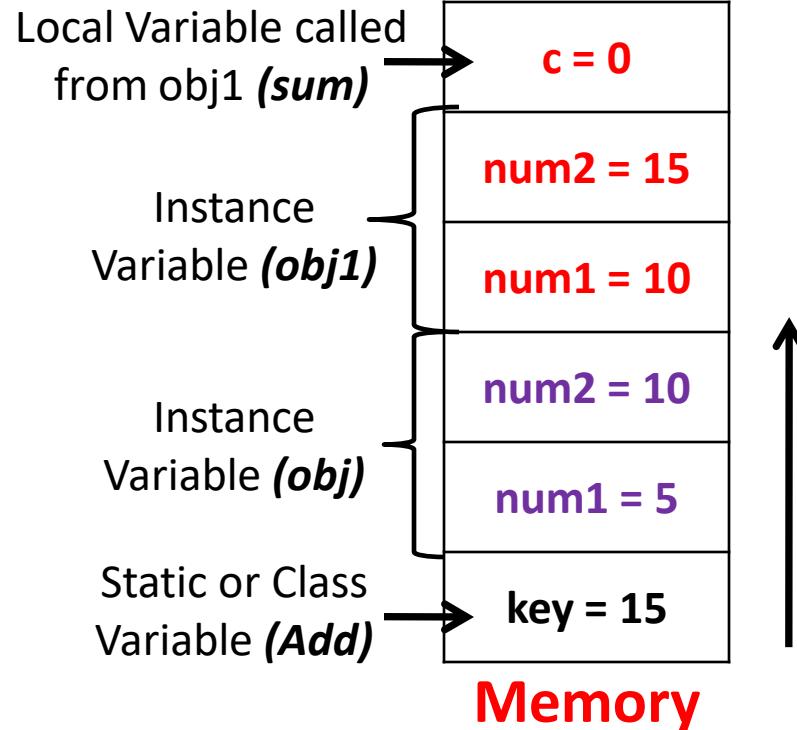
- Public -> Members are accessible from outside the class
- Private -> Members cannot be accessed (or viewed) from outside the class
- Protected -> Members cannot be accessed from outside the class, however, they can be accessed in inherited classes

Miscellaneous

```

class Add
{
public:
    int num1, num2;           → Instance Variables
    static int key; // Default value = 0 → Static or Class Variable
    int sum(int n1, int n2)
    {
        int c = 0;           → Local Variable
        return n1 + n2;
    }
}
int main()
{
    //Creating object of class
    Add obj, obj1;
    obj.num1 = 5;  obj1.num1 = 10;
    obj.num2 = 10; obj1.num2 = 15;
    cout << obj1.sum(num1, num2);
}
int Add::key = 15;

```





Miscellaneous

```
//Base class  
class Parent  
{  
    public:  
        int id_p;  
};  
  
class Child : public Parent  
{  
    public:  
        int id_c;  
};
```

```
int main()  
{  
    Child obj1;  
    obj1.id_c = 7;  
    obj1.id_p = 91;  
    cout << "Child id is " << obj1.id_c <<  
    endl;  
    cout << "Parent id is " << obj1.id_p <<  
    endl;  
  
    return 0;  
}
```

Inheritance -> Is Unidirectional

Types

- Single Inheritance
- Multiple Inheritance

O/P: Child id is 7
Parent id is 91



Miscellaneous

```
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler
{
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle"
        << endl;
    }
};
```

```
class Car: public Vehicle, public
FourWheeler {

};

int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Inheritance

Types

- Single Inheritance
- Multiple Inheritance

O/P: This is a Vehicle
This is a 4 wheeler Vehicle



Miscellaneous

```
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler
{
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle"
        << endl;
    }
};
```

```
class Car: public Vehicle, public
FourWheeler {
public:
    Car()
    {
        cout << "This is a Car" << endl;
    }
};

int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

O/P: This is a Vehicle
This is a 4 wheeler Vehicle
This is a Car

Inheritance

Types

- Single Inheritance
- Multiple Inheritance



Miscellaneous

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};
```

```
class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};
```

Inheritance

```
class D : private A // 'private' is default for
classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)



Miscellaneous

```
class Geeks
{
public:
void func(int x)
{
    cout << "value of x is " << x <<
endl;
}

void func(double x)
{
    cout << "value of x is " << x <<
endl;
}

void func(int x, int y)
{
    cout << "value of x and y is " << x
<< ", " << y << endl;
}
};
```

```
int main()
{
    Geeks obj1;
    obj1.func(7);
    obj1.func(9.132);
    obj1.func(85, 64);
    return 0;
}
```

Polymorphism

When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments

Types

- Functional Overloading (Compile Time)
- Operator Overloading (Compile Time)
- Virtual Functions (Run Time)



Miscellaneous

- **Function Overloading (achieved at compile time)**
 - Provides multiple definitions of the function by changing signature
 - That is, changing number of parameters, change datatype of parameters (Return type doesn't play any role)
 - It can be done in base as well as derived class
- **Example:**

```
void area(int a);
void area(int a, int b);
```
- **Function Overriding (achieved at run time)**
 - Redefinition of base class function in its derived class with same signature
 - That is, return type and parameters
 - It can only be done in derived class

Function Overloading vs Function Overriding

Example:

```
Class a
{
    public: virtual void display()
    {
        cout << "hello";
    }
};

Class b:public a
{
    public: void display()
    {
        cout << "bye";
    }
};
```

<https://www.geeksforgeeks.org/function-overloading-vs-function-overriding-in-cpp/>



Miscellaneous

```
class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
    Complex operator + (Complex const
&obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print()
    {
        cout << real << " + " << imag << endl;
    }
};
```

```
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example
    call to "operator+"
    c3.print();
}
```

“+” -> Addition (Two Integers),
Concatenation (Strings), **Addition
(Complex Numbers)**

Types

- Functional Overloading (Compile Time)
- **Operator Overloading (Compile Time)**
- Virtual Functions (Run Time)



Miscellaneous

```
class base
{
public:
    virtual void print ()
    {
        cout<< "print base class" << endl;
    }

→ void show ()
{
    cout<< "show base class" << endl;
}
};

class derived: public base
{
public:
    → void print () //print () is already virtual
function in derived class, we could also
declared as virtual void print () explicitly
    {
        cout<< "print derived class" << endl;
    }

void show ()
{
    cout<< "show derived class" << endl;
}

O/P: print derived class
show base class
```

```
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
    return 0;
}
```

<https://hownot2code.com/2017/08/10/c-pointers-why-we-need-them-when-we-use-them-how-they-differ-from-accessing-to-object-itself/>

This type of polymorphism is achieved by Function Overriding. Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden

Types

- Functional Overloading (Compile Time)
- Operator Overloading (Compile Time)
- **Virtual Functions (Run Time)**



Miscellaneous

```
class base
{
public:
    → void print ()
    {
        cout<< "print base class" << endl;
    }

    → void show ()
    {
        cout<< "show base class" << endl;
    }
};

class derived: public base
{
public:
    void print ()
    {
        cout<< "print derived class" << endl;
    }

    void show ()
    {
        cout<< "show derived class" << endl;
    }
};
```

```
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //Non-virtual function, binded at compile time
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}
```

Note: “Virtual” keyword is not used. Hence, objects are binded at compile time itself

O/P: print base class
show base class



Miscellaneous

```
class base
{
public:
    void print ()
    {
        cout<< "print base class" << endl;
    }

    void show ()
    {
        cout<< "show base class" << endl;
    }
};

class derived: public base
{
public:
    void print ()
    {
        cout<< "print derived class" << endl;
    }

    void show ()
    {
        cout<< "show derived class" << endl;
    }
};
```

```
int main()
{
    base bptr;
    bptr.print();
    bptr.show();
    derived d;
    d.print();
    d.show();

    return 0;
}
```

Note: Individual objects are created

O/P: print base class
show base class
print derived class
show derived class



Miscellaneous

```
class base
{
public:
    virtual void print ()
    {
        cout<< "print base class" << endl;
    }

    void show ()
    {
        cout<< "show base class" << endl;
    }
};

class derived: public base
{
public:
    void print () //print () is already virtual
    function in derived class, we could also
    declared as virtual void print () explicitly
    {
        cout<< "print derived class" << endl;
    }

    void show ()
    {
        cout<< "show derived class" << endl;
    }
};
```

```
int main()
{
    base bptr;
    bptr.print();
    bptr.show();
    derived d;
    d.print();
    d.show();

    return 0;
}
```

Note: In the case of individual objects, there is no effect even though virtual keyword is added

O/P: print base class
show base class
print derived class
show derived class



Miscellaneous

```
class Base
{
    int x;
public:
    virtual void fun() = 0; // Pure Virtual Function
    int getX() { return x; }
};

// This class inherits from Base and implements
fun()
class Derived: public Base
{
    int y;
public:
    void fun()
    {
        cout << "fun() called";
    }
};
int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

Abstract Method or Pure Virtual Method in C++

virtual void fun() = 0; // Pure Virtual Function

class Test

```
{  
    int x;  
public:
```

virtual void show() = 0;
int getX() { return x; }

```
};
```

int main(void)

```
{  
    Test t;  
    return 0;  
}
```

Abstract Class (Interface) or Abstract Base Class in C++

Such a class usually cannot be instantiated, because some of its methods are declared but are not defined (they do not have bodies). Any subclass of an abstract class that is to be instantiated must provide implementations (definitions) of all of the inherited abstract methods.

O/P: fun() called

O/P: Compiler Error: cannot declare variable 't' to be of abstract type 'Test' because the following virtual functions are pure within 'Test': note: virtual void Test::show()



Miscellaneous

```
class Base
{
    int x;
public:
    virtual void fun() = 0; //Pure Virtual Function
    virtual void fun1() = 0; // Pure Virtual Function
    int getX()
    {
        return x;
    }
};

class Derived: public Base
{
    int y;
public:
    virtual void fun() = 0;
    virtual void fun1() = 0;
    virtual void fun2() = 0;
};
```

```
Class Derived_1: public Derived
{
    int z:
public:
    virtual void fun()
    {
        cout << "Show";
    }
    virtual void fun1()
    {
        cout << "Show_1";
    }
    virtual void fun2()
    {
        cout << "Show_2";
    }
};

int main(void)
{
    Base b; // Throws Error
    Derived d; // Throws Error
    Derived_1 d;
    d.fun();
    return 0;
}
```

O/P: Show



Miscellaneous

```
class MyClass {      // The class
    public:        // Access specifier
        void myMethod(); // Method/function declaration
};

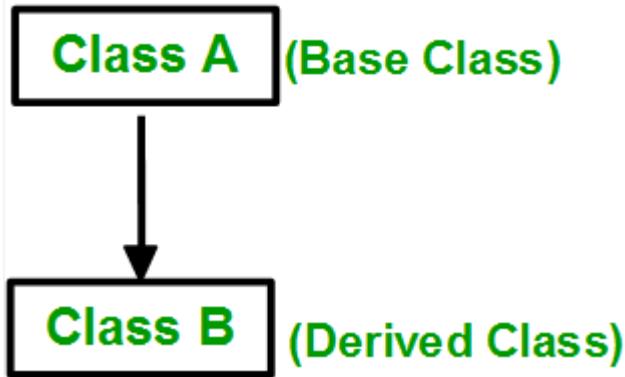
// Method/function definition outside the class
void MyClass::myMethod() {
    cout << "Hello World!";
}

int main() {
    MyClass myObj;    // Create an object of MyClass
    myObj.myMethod(); // Call the method
    return 0;
}
```

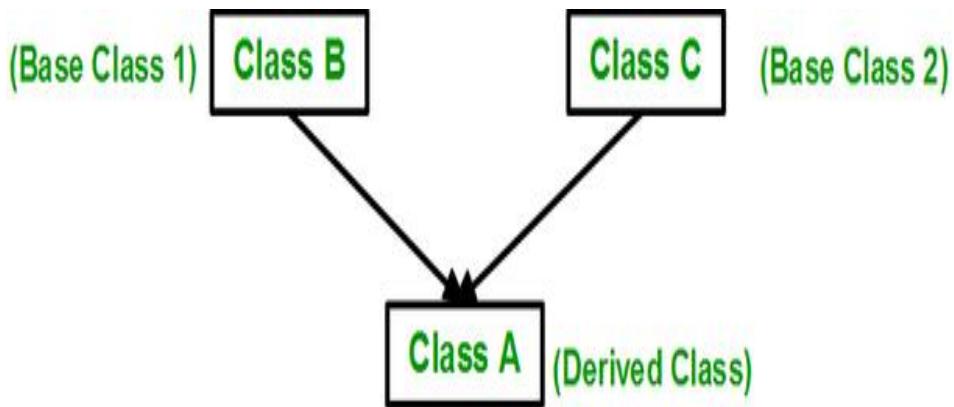


Miscellaneous

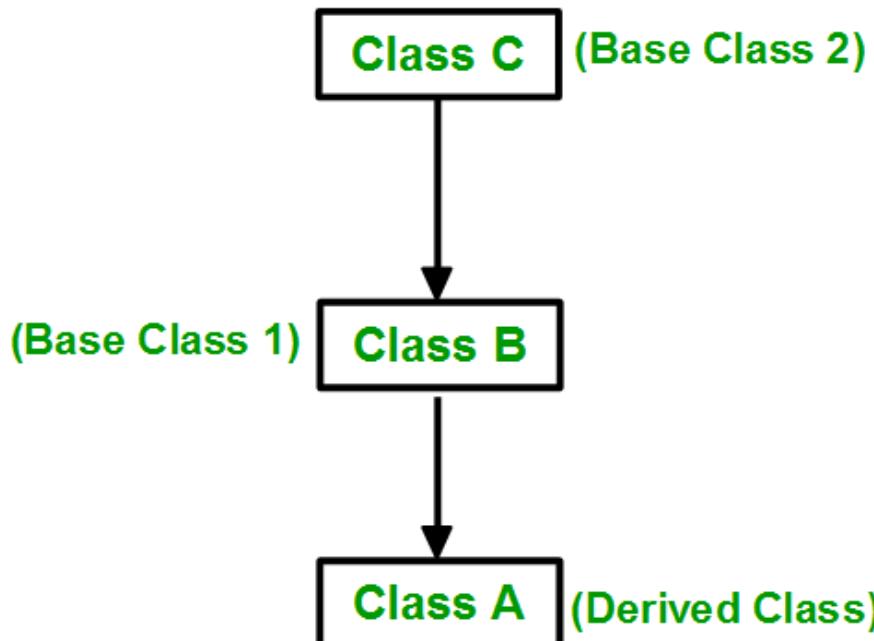
Single Inheritance



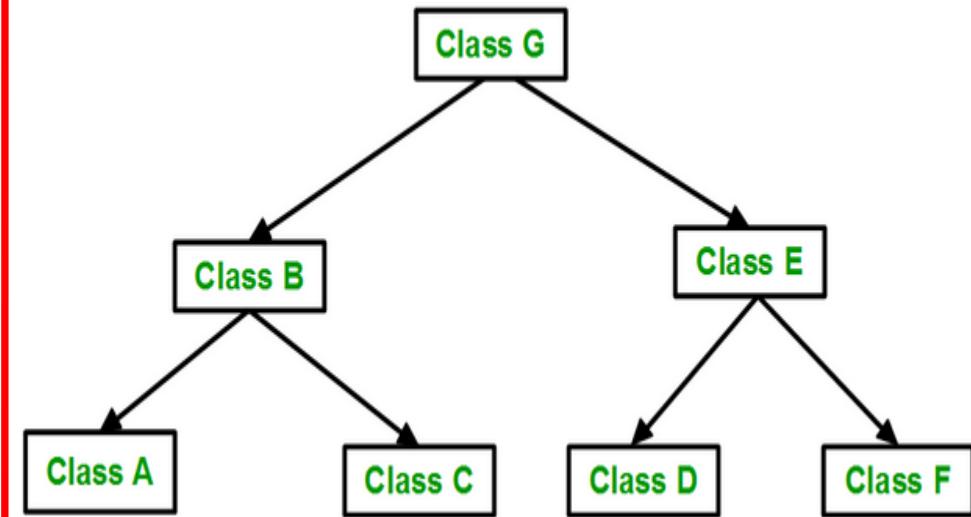
Multiple Inheritance



Multi-Level Inheritance

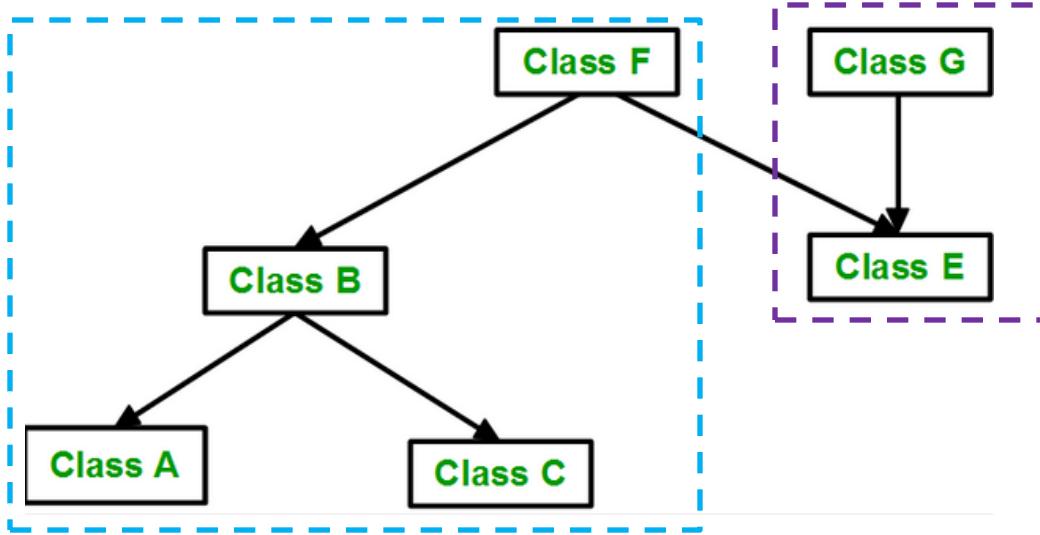


Hierarchical Inheritance

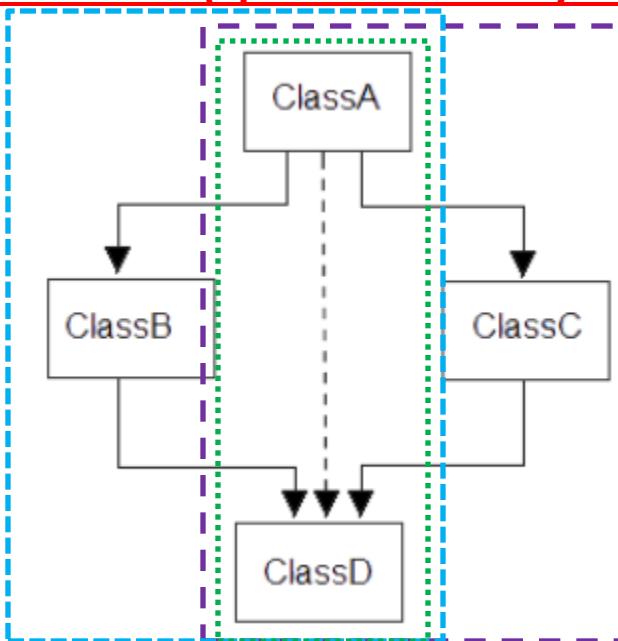


Miscellaneous

Hybrid (Virtual) Inheritance



Multipath Inheritance (Special Case of Hybrid Inheritance)





Miscellaneous

```
class Node
{
    private:
        int key;
        Node* next;
    /* Other members of Node Class */

    friend class LinkedList;
    // Now class LinkedList can
    // access private members of Node
};

Class LinkedList
{
    public:
        int search()
    {
        ....
    }
};
```

Friend Class

class Node

```
{
```

```
private:
```

```
int key;
```

```
Node* next;
```

```
/* Other members of Node Class */
```

Friend Function

```
private:
```

```
int key;
```

```
Node* next;
```

```
/* Other members of Node Class */
```

```
friend int LinkedList::search();
```

```
// Only search() of linkedList
// can access internal members
```

```
};
```

- A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class
- Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:
 - a) A method of another class
 - b) A global function



Miscellaneous

```
class base
{
public:
    base()
    {
        cout << "Base Class Constructor Called"
        << endl;
    }
    void print ()
    {
        cout << "print base class" << endl;
    }
    void show ()
    {
        cout << "show base class" << endl;
    }
~base()
{
    cout << "Base Class Destructor Called"
    << endl;
}
```

class derived: public base

```
{
public:
    derived()
    {
        cout << "Dervied Class Destructor
Called" << endl;
    }
    void print ()
    {
        cout << "print derived class" << endl;
    }
    void show ()
    {
        cout << "show derived class" << endl;
    }
~derived()
{
    cout << "Derived Class Destructor
Called" << endl;
}
```

Destructor

<https://hownot2code.com/2017/08/10/c-pointers-why-we-need-them-when-we-use-them-how-they-differ-from-accessing-to-object-itself/>



Miscellaneous

```
int main()
{
    base bptr;
    bptr.print();
    bptr.show();
    derived d;
    d.print();
    d.show();
    cout << "1" << endl;
→ return 0;
    cout << "2" << endl;
}
```

O/P: Base Class Constructor Called

print base class
show base class
Base Class Constructor Called
Derived Class Constructor Called
print derived class
show derived class

1

Object "d" {
 deleted } Derived Class Destructor Called
Base Class Destructor Called
Base Class Destructor Called

```
int main()
{
    base *bptr;
    derived d;
bptr = &d;
    bptr->print();
    bptr->show();
    cout << "1" << endl;
→ return 0;
}
```

Destructor

O/P: Base Class Constructor Called
Derived Class Constructor Called
print base class
show base class
1
Derived Class Destructor Called
Base Class Destructor Called

Object "bptr"
→ deleted



Miscellaneous

```
int main()
{
    base *bptra1 = new base();
    base *bptra;
    derived d;
    bptra = &d;
    bptra->print();
    bptra->show();
    cout << "1" << endl;
    → return 0;
}
```

O/P: Base Class Constructor Called

Base Class Constructor Called
Derived Class Constructor Called
print base class
show base class

1

Object "d" deleted } Derived Class Destructor Called
Base Class Destructor Called

Note: Object "bptra1" is not deleted

```
int main()
{
    base *bptra1 = new base();
    delete bptra1;
    base *bptra;
    derived d;
    bptra = &d;
    bptra->print();
    bptra->show();
    cout << "1" << endl;
    → return 0;
}
```

Destructor

Note: Whenever you create an object in heap memory using "new" keyword, you have to delete it explicitly using "delete" keyword. Here, object "bptra1" is deleted explicitly

O/P: Base Class Constructor Called

Base Class Destructor Called } Object
"bptra1" deleted

Base Class Constructor Called

Derived Class Constructor Called

print base class

show base class

1

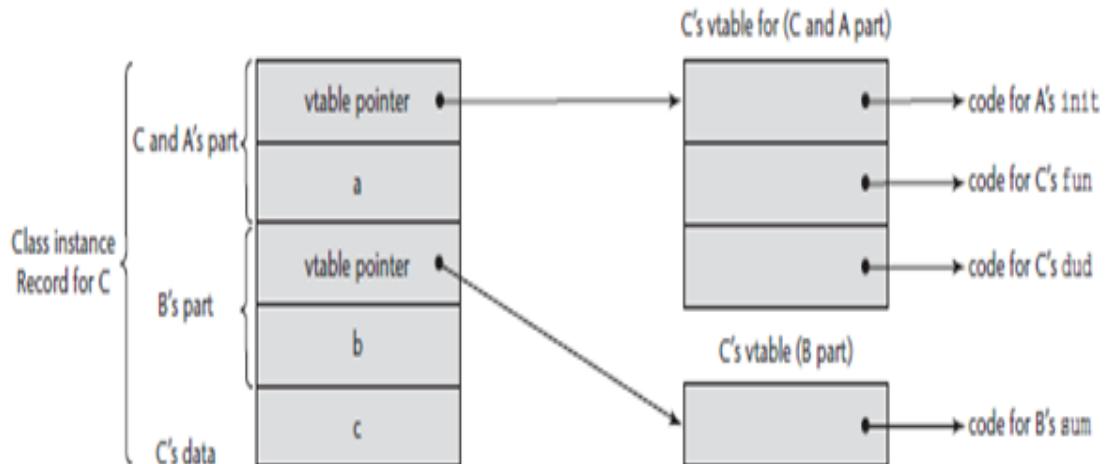
Object "d" deleted } Derived Class Destructor Called
Base Class Destructor Called

Miscellaneous

```
int main()
{
    base *bptr1;
    bptr1->print();
    bptr1->show();
    cout << "1" << endl;
    return 0;
}
```

O/P: print base class
show base class
1

Note: Neither constructor
nor destructor are called in
this case





Introduction

- All data are objects
- A language that is object oriented must provide support for three key language features -> Abstract Data Types, Inheritance and Dynamic Binding of method calls to methods

Inheritance

- Software reuse
- Abstract data types, with their encapsulation and access controls, are obviously candidates for reuse
 - The problem with the reuse of abstract data types is that, in nearly all cases, the features and capabilities of the existing type are not quite right for the new use
 - A second problem with programming with abstract data types is that the type definitions are all independent and are at the same level



Introduction

- Inheritance offers a solution to both the modification problem posed by abstract data type reuse and the program organization problem
- Programmers can begin with an existing abstract data type and design a modified descendant of it to fit a new problem requirement
- Abstract data types in object-oriented languages -> Classes
- Instances of abstract data types (class instances) -> Objects
- A class that is defined through inheritance from another class -> Derived Class or Subclass
- A class from which the new class is derived is its parent class or superclass
- Subprograms that define the operations on objects of a class are called methods
- The calls to methods are sometimes called messages
- The entire collection of methods of an object is called the message protocol or message interface of the object

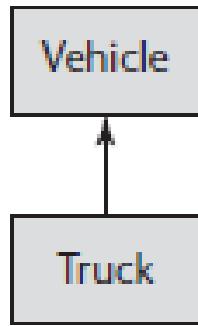


Introduction

- Computations in an object-oriented program are specified by messages sent from objects to other objects, or in some cases, to classes
- Passing a message is indeed different from calling a subprogram
 - A subprogram typically processes data that is either passed by its caller as a parameter or is accessed nonlocally or globally
 - A subprogram defines a process that it can perform on any data sent to it (or made available nonlocally or globally)
 - A message sent to an object is a request to execute one of its methods
 - At least part of the data on which the method is to operate is the object itself
 - Because the objects are of abstract data types, these should be the only ways to manipulate the object



Introduction



Variables -> Year, Color, Make

Variables -> Year, Color, Make, Capacity, Number of Wheels

- Several ways a derived class can differ from its parent
 - Parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass
 - Subclass can add variables and/or methods to those inherited from the parent class
 - Subclass can modify the behavior of one or more of its inherited methods
 - Modified method has the same name, and often the same protocol, as the one of which it is a modification
- New method is said to override the inherited method, which is then called an overridden method
- Purpose of an overriding method is to provide an operation in the subclass that is similar to one in the parent class, but is customized for objects of the subclass. **Eg:** Draw() method in Class Rectangle (Length, Width), Circle (Radius)



Introduction

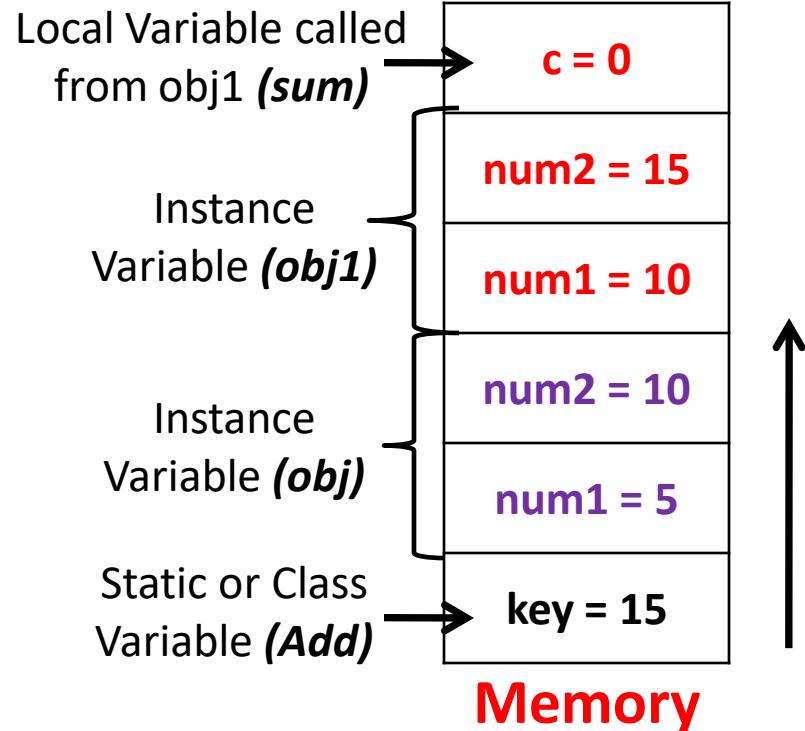
- Classes can have two kinds of methods and two kinds of variables
- Commonly used methods and variables are called instance methods and instance variables
- Every object of a class has its own set of instance variables, which store the object's state
- Only difference between two objects of the same class is the state of their instance variables
- **Eg:** A class for cars might have instance variables for color, make, model and year
- Instance methods operate only on the objects of the class
- Class variables belong to the class, rather than its object, so there is only one copy for the class
- **Eg:** Counter to count the number of instances of a class -> The counter would need to be a class variable
- Class methods can perform operations on the class and possibly also on the objects of the class

Miscellaneous

```

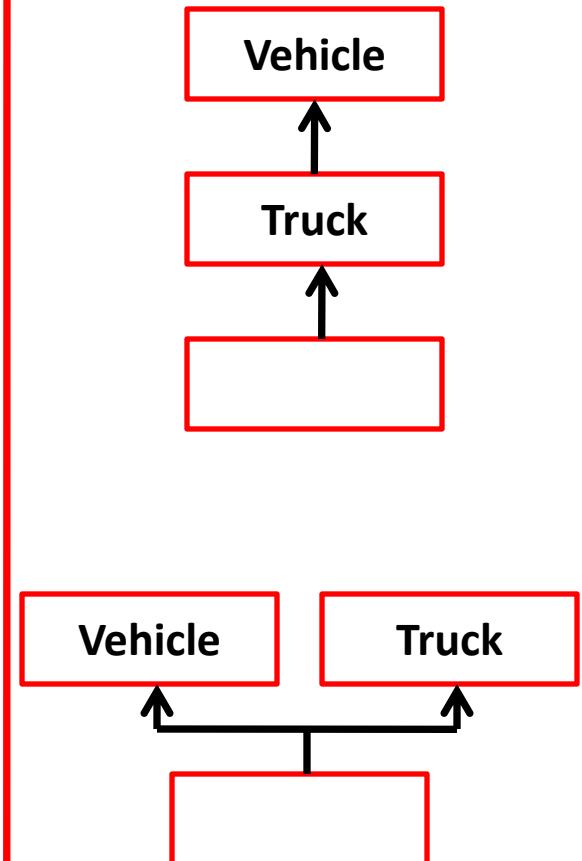
class Add
{
public:
    int num1, num2;           → Instance Variables
    static int key; // Default value = 0 → Static or Class Variable
    int sum(int n1, int n2)
    {
        int c = 0;           → Local Variable
        return n1 + n2;
    }
}
int main()
{
    //Creating object of class
    Add obj, obj1;
    obj.num1 = 5;  obj1.num1 = 10;
    obj.num2 = 10; obj1.num2 = 15;
    cout << obj1.sum(num1, num2);
}
int Add::key = 15;

```



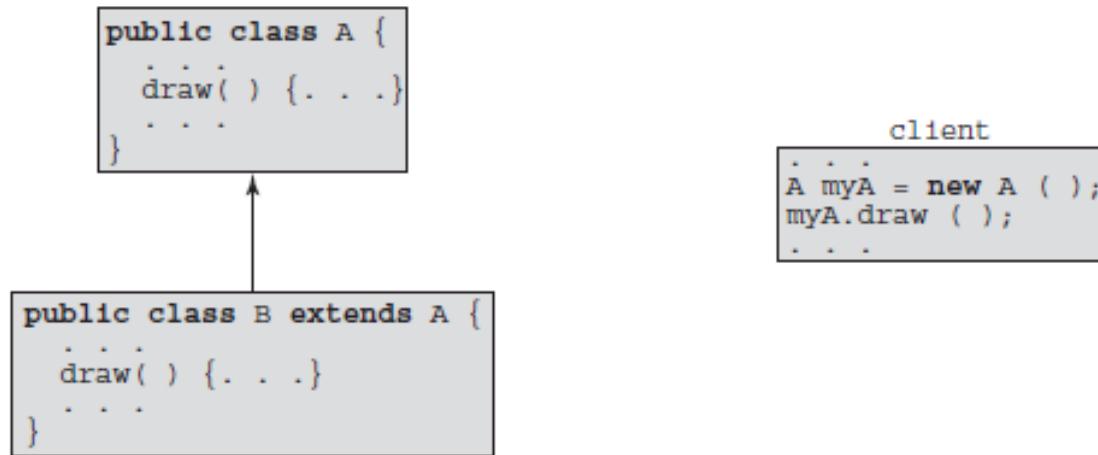
Introduction

- If a new class is a subclass of a single parent class, then the derivation process is called single inheritance
- If a class has more than one parent class, the process is called multiple inheritance
- When a number of classes are related through single inheritance, their relationships to each other can be shown in a derivation tree
- The class relationships in a multiple inheritance can be shown in a derivation graph
- One disadvantage of inheritance is that it creates dependencies among the classes in an inheritance hierarchy
- This result works against one of the advantages of abstract data types, which is that they are independent of each other
- Of course, not all abstract data types must be completely independent
- But in general, the independence of abstract data types is one of their strongest positive characteristics
- However, it may be difficult, if not impossible, to increase the reusability of abstract data types without creating dependencies among some of them
- Furthermore, in many cases, the dependencies naturally mirror dependencies in the underlying problem space



Dynamic Binding

- Third characteristic (after abstract data types and inheritance) of object-oriented programming languages is a kind of polymorphism provided by the dynamic binding of messages to method definitions -> Dynamic Dispatch



- If a client of A and B has a variable that is a reference to class A's objects, that reference also could point at class B's objects, making it a polymorphic reference
- If the method draw, which is defined in both classes, is called through the polymorphic reference, the run-time system must determine, during execution, which method should be called, A's or B's (by determining which type object is currently referenced by the reference)



Miscellaneous

```
#include <iostream>
class Point
{
    int x, y, z;
public:
    Point(int x, int y, int z)
    {
        this->x = x;
        this->y = y;
        this->z = z;
    }
    void display()
    {
        cout << "(" << x << ", " << y << ", "
             << z << ")" << endl;
    }
};
```

```
int main()
{
    Point p1(10, 15, 20);
    p1.display();
    Point *ptr;
    ptr = new Point(50, 60, 70);
    ptr->display();
}
```

O/P: (10, 15, 20)
(50, 60, 70)

- In C++, we can instantiate the class object with or without using the new keyword. If the new keyword is not used, then it is like normal object. This will be stored at the stack section. This will be destroyed when the scope ends. But for the case when we want to allocate the space for the item dynamically, then we can create pointer of that class, and instantiate using new operator.
- In C++, the new is used to dynamically allocate memory.



Miscellaneous

```
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
    virtual void draw()
    {
        cout<<"Vehicle Draw";
    }
};
```

O/P: This is a Vehicle
This is a 4 wheeler Vehicle
Fourwheeler Draw

```
class FourWheeler: public Vehicle
{
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler
Vehicle" << endl;
    }
    void draw()
    {
        cout<<"Fourwheeler Draw";
    }
};

void main()
{
    Vehicle *obj = new FourWheeler();
    obj->draw();
}
```

FourWheeler *obj = new Vehicle();
O/P: error: invalid conversion from
'Vehicle*' to 'FourWheeler*' [-fpermissive]



Dynamic Binding

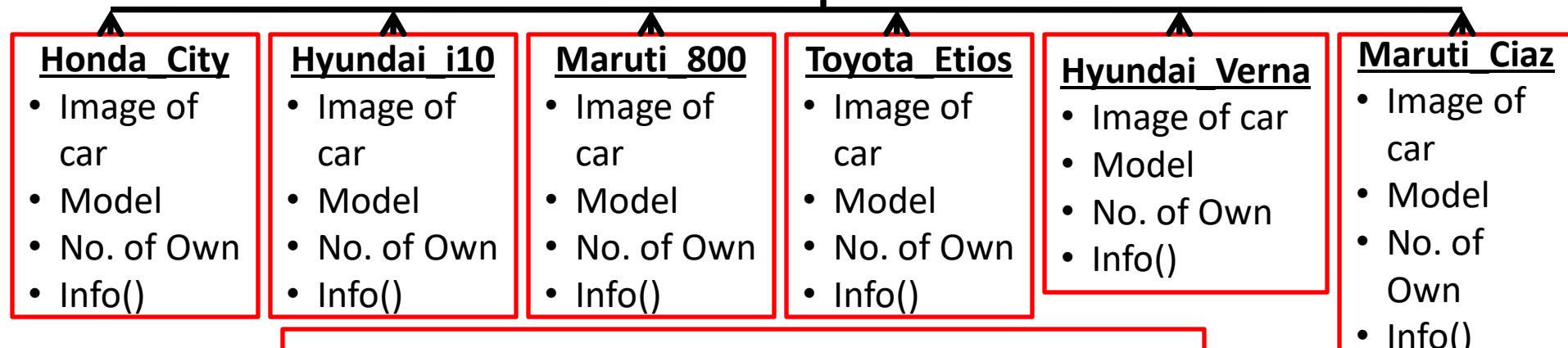
- Polymorphism makes a statically typed language a little bit dynamically typed
 - Little bit is in some bindings of method calls to methods
- The type of a polymorphic variable is indeed dynamic
- One purpose of dynamic binding is to allow software systems to be more easily extended during both development and maintenance

Suppose we have a catalog of used cars that is implemented as a car class and a subclass for each car in the catalog. The subclasses contain an image of the car and specific information about the car. Users can browse the cars with a program that displays the images and information about each car as the user browses to it. The display of each car (and its information) includes a button that the user can click if he or she is interested in that particular car. After going through the whole catalog, or as much of the catalog as the user wants to see, the system will print the images and information about the cars of interest to the user. One way to implement this system is to place a reference to the object of each car of interest in an array of references to the base class, car. When the user is ready, information about all of the cars of interest could be printed for the user to study and compare the cars in the list. The list of cars will of course change frequently. This will necessitate corresponding changes in the subclasses of car. However, changes to the collection of subclasses will not require any other changes to the system.



Dynamic Binding

Suppose we have a catalog of used cars that is implemented as a **car** class and a subclass for each car in the catalog. The subclasses contain an image of the car and specific information about the car. Users can browse the cars with a program that displays the images and information about each car as the user browses to it. The display of each car (and its information) includes a button that the user can click if he or she is interested in that particular car. After going through the whole catalog, or as much of the catalog as the user wants to see, the system will print the images and information about the cars of interest to the user.



```
Maruti_800 mar = new Maruti_800();
Honda_City hon = new Honda_City();
Hyundai_i10 hyn = new Hyundai_i10(); car *obj[];
obj[0] = mar; obj[1] = hon; obj[2] = hyn;
```

Design Issues for Object-Oriented Languages



- A number of issues must be considered when designing the programming language features to support inheritance and dynamic binding

Exclusivity of Objects

- A language designer who is totally committed to the object model of computation designs an object system that subsumes all other concepts of type
- Everything, from a simple scalar integer to a complete software system, is an object in this mind-set
- Advantage of this choice is the elegance and pure uniformity of the language and its use
- Disadvantage is that simple operations must be done through the message-passing process, which often makes them slower than similar operations in an imperative model, where single machine instructions implement such simple operations **// ADD B, A**
- In this purest model of object-oriented computation, all types are classes
- There is no distinction between predefined and user-defined classes
- All classes are treated the same way and all computation is accomplished through message passing

Design Issues for Object-Oriented Languages



- One alternative to the exclusive use of objects that is common in imperative languages to which support for object-oriented programming has been added is to retain the complete collection of types from a traditional imperative programming language and simply add the object typing model
 - Results in a larger language whose type structure can be confusing to all but expert users
- Another alternative to the exclusive use of objects is to have an imperative style type structure for the primitive scalar types, but implement all structured types as objects
 - Provides the speed of operations on primitive values that is comparable to those expected in the imperative model
 - Also leads to complications in the language
 - Nonobject values must be mixed with objects
 - Creates a need for so-called *wrapper classes* for the nonobject types, so that some commonly needed operations can be implemented as methods of the wrapper class

Design Issues for Object-Oriented Languages



Are Subclasses Subtypes?

- Issue here is relatively simple: Does an “is-a” relationship hold between a derived class and its parent class?
- If a derived class is a parent class, then objects of the derived class must expose all of the members that are exposed by objects of the parent class
- At a less abstract level, an is-a relationship guarantees that in a client a variable of the derived class type could appear anywhere a variable of the parent class type was legal, without causing a type error
- Moreover, the derived class objects should be behaviorally equivalent to the parent class objects
- Ada -> subtype **Small_Int** is **Integer** range **-100, ..., 0, ..., 100;**
 - Every **Small_Int** variable is, in a sense, an **Integer** variable
 - Variables of **Small_Int** type have all of the operations of **Integer** variables but can store only a subset of the values possible in **Integer**
 - Furthermore, every **Small_Int** variable can be used anywhere an **Integer** variable can be used
 - That is, every **Small_Int** variable is, in a sense, an **Integer** variable



Miscellaneous

```
//Base class  
class Parent  
{  
    public:  
        int id_p;  
};  
  
class Child : public Parent  
{  
    public:  
        int id_c;  
};  
  
class child_derived: public Child  
{  
};
```

```
int main()  
{  
    Parent obj;  
    obj.id_p = 9;  
    Child obj1;  
    obj1.id_c = 7;  
    obj1.id_p = 91;  
    cout << "Parent id is " << obj.id_p <<  
    endl;  
    cout << "Child id is " << obj1.id_p <<  
    endl;  
  
    return 0;  
}
```

O/P: Parent id is 9
Child id is 91

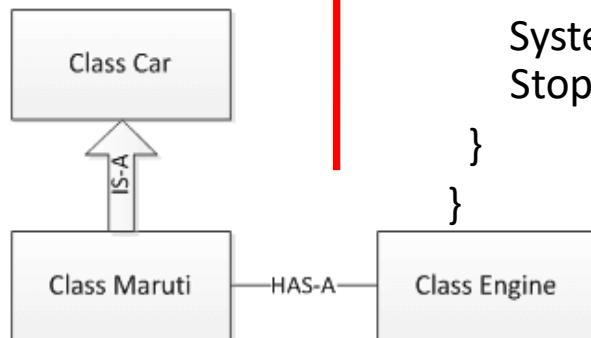
Inheritance -> Is
Unidirectional

Types

- Single Inheritance
- Multiple Inheritance

Java:

```
class Car {
    private String color;
    private int maxSpeed;
    public void carInfo()
    {
        System.out.println("Car Color= "+color
            + " Max Speed= " + maxSpeed);
    }
    public void setColor(String color)
    {
        this.color = color;
    }
    public void setMaxSpeed(int maxSpeed)
    {
        this.maxSpeed = maxSpeed;
    }
}
```



Is-A vs Has-A

```
class Maruti extends Car
{
    public void MarutiStartDemo()
    {
        Engine MarutiEngine = new Engine();
        MarutiEngine.start();
    }
}
public class Engine
{
    public void start()
    {
        System.out.println("Engine Started:");
    }
    public void stop()
    {
        System.out.println("Engine Stopped:");
    }
}
```

Composition (HAS-A) simply mean the use of instance variables that are references to other objects



Miscellaneous

```
public class RelationsDemo
{
    public static void main(String[] args)
    {
        Maruti myMaruti = new Maruti();
        myMaruti.setColor("RED");
        myMaruti.setMaxSpeed(180);
        myMaruti.carInfo();
        myMaruti.MarutiStartDemo();
    }
}
```

O/P:

**Car Color= RED Max Speed= 180
Engine Started:**



Miscellaneous

- In OOP, IS-A relationship is completely inheritance
- This means, that the child class is a type of parent class. Eg: An apple is a fruit. So you will extend fruit to get apple
- Composition means creating instances which have references to other objects
- Eg: A room has a table. So you will create a class room and then in that class create an instance of type table

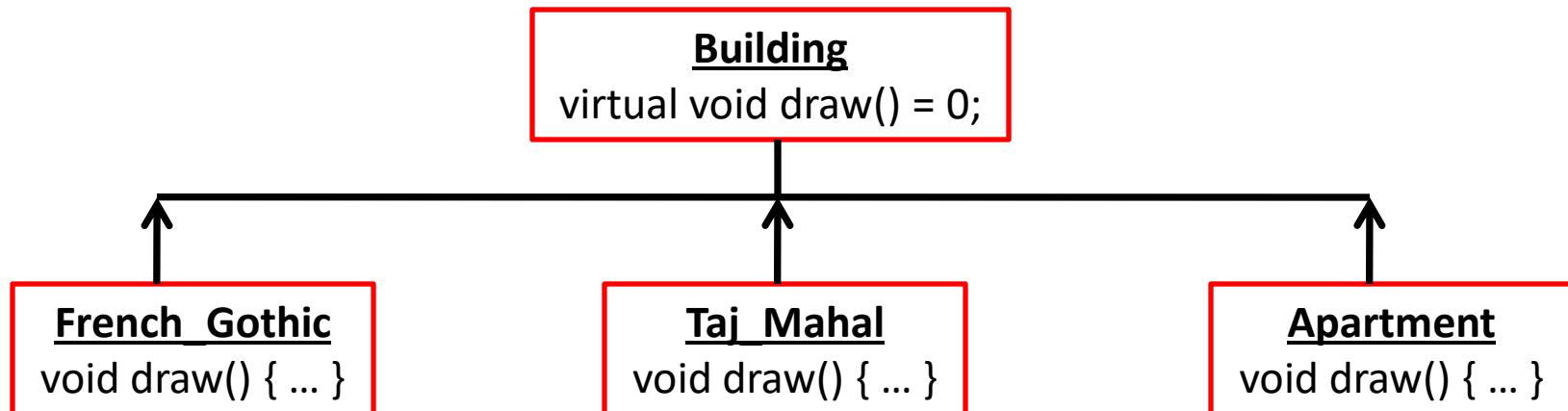
```
class Room {  
    Table table = new Table();  
}
```

- A HAS-A relationship is dynamic (run time) binding while inheritance is a static (compile time) binding
- If you just want to reuse the code and you know that the two are not of same kind use composition
- Eg: You cannot inherit an oven from a kitchen. A kitchen HAS-A oven.
- When you feel there is a natural relationship like Apple is a Fruit use inheritance

Miscellaneous

Need for Virtual Function:

- Suppose a program defined a Building class and a collection of subclasses for specific types of buildings, for instance, French_Gothic
- It probably would not make sense to have an implemented draw method in Building
- But because all of its descendant classes should have such an implemented method, the protocol (but not the body) of that method is included in Building



Design Issues for Object-Oriented Languages



- There are a wide variety of ways in which a subclass could differ from its base or parent class
 - Subclass could have additional methods
 - Could have fewer methods
 - Types of some of the parameters could be different in one or more methods
 - **Return type of some method could be different**
 - Number of parameters of some method could be different
 - **Body of one or more of the methods could be different**
- Most programming languages severely restrict the ways in which a subclass can differ from its base class
- In most cases, the language rules restrict the subclass to be a subtype of its parent class
- A derived class is called a subtype if it has an is-a relationship (derived using public access specifier) with its parent class

Design Issues for Object-Oriented Languages



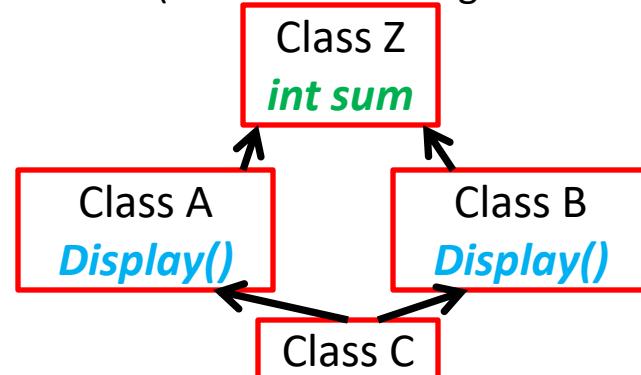
- Characteristics of a subclass that ensure that it is a subtype are as follows:
 - Methods of the subclass that override parent class methods must be type compatible with their corresponding overridden methods
 - Compatible here means that a call to an overriding method can replace any call to the overridden method in any appearance in the client program without causing type errors
 - That means that every overriding method must have the same number of parameters as the overridden method and the types of the parameters and the return type must be compatible with those of the parent class
 - Having an identical number of parameters and identical parameter types and return type would, of course, guarantee compliance of a method
- Derivation process for subtypes must require that public entities of the parent class are inherited as public entities in the subclass

Design Issues for Object-Oriented Languages



Single and Multiple Inheritance

- Another simple issue is: Does the language allow multiple inheritance (in addition to single inheritance)?
- Reasons lie in two categories: Complexity and Efficiency
- Additional complexity is illustrated by several problems

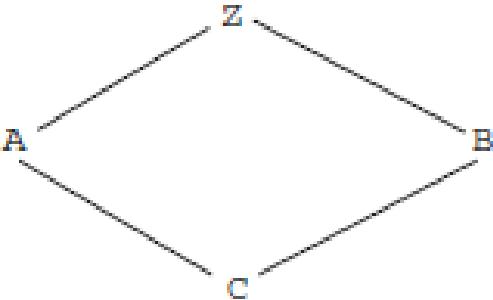


- If C needs to reference both versions of display, how can that be done?
- This ambiguity problem is further complicated when the two parent classes both define identically named methods and one or both of them must be overridden in the subclass
- Another issue arises if both A and B are derived from a common parent Z, and C has both A and B as parent classes -> Diamond or Shared Inheritance
- In this case, both A and B should include Z's inheritable variables
- Suppose Z includes an inheritable variable named "sum"
- The question is whether C should inherit both versions of sum or just one, and if just one, which one?
- There may be programming situations in which just one of the two should be inherited, and others in which both should be inherited

Design Issues for Object-Oriented Languages



- The question of efficiency may be more perceived than real
- In C++, for example, supporting multiple inheritance requires just one additional array access and one extra addition operation for each dynamically bound method call
- Although this operation is required even if the program does not use multiple inheritance, it is a small additional cost



- Interfaces (abstract class) are an alternative to multiple inheritance
- Interfaces provide some of the benefits of multiple inheritance but have fewer disadvantages

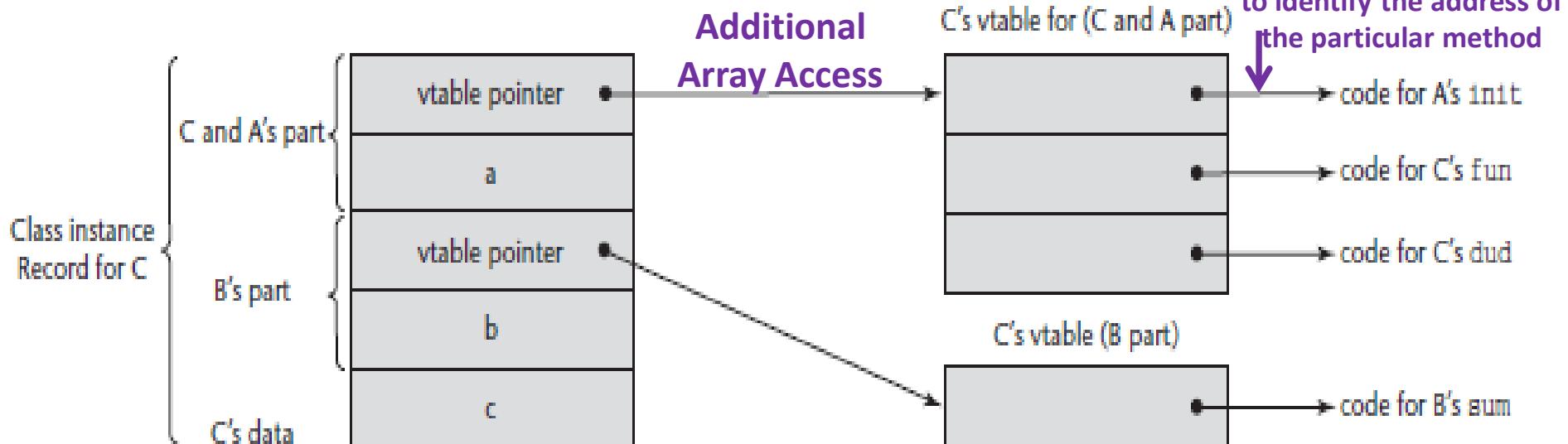
Miscellaneous

```
class A {
public:
    int a;
    virtual void fun() { ... }
    virtual void init() { ... }
};

class B {
```

```
public:
    int b;
    virtual void sum() { ... }
};

class C : public A, public B {
public:
    int c;
    virtual void fun() { ... }
    virtual void dud() { ... }
};
```



Design Issues for Object-Oriented Languages



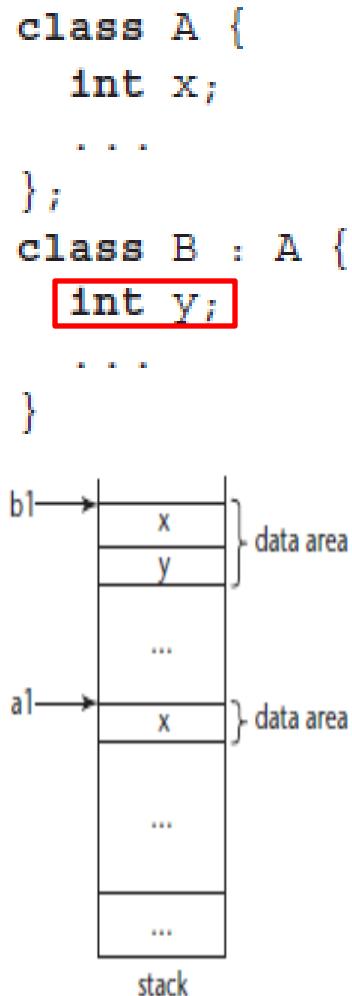
Allocation and Deallocation of Objects

- Two design questions concerning the allocation and deallocation of objects
 - First question is the place from which objects are allocated
 - If they behave like the abstract data types, then perhaps they can be allocated from anywhere
 - This means they could be allocated from the run-time stack or explicitly created on the heap with an operator or function, such as new
 - If they are all heap dynamic, there is the advantage of having a uniform method of creation and access through pointer or reference variables
 - This design simplifies the assignment operation for objects, making it in all cases only a pointer or reference value change
 - Also allows references to objects to be implicitly dereferenced, simplifying the access syntax

Design Issues for Object-Oriented Languages



- If objects are stack dynamic (value variables), there is a problem with regard to subtypes
 - If class B is a child of class A and B is a subtype of A, then an object of B type can be assigned to a variable of A type
 - **Eg:** If b1 is a variable of B type and a1 is a variable of A type, then **a1 = b1;** is a legal statement
 - If a1 and b1 are references to heap-dynamic objects, there is no problem—the assignment is a simple pointer assignment
 - However, if a1 and b1 are stack dynamic, then they are value variables and, if assigned the value of the object, must be copied to the space of the target object
 - If B adds a data field to what it inherited from A, then a1 will not have sufficient space on the stack for all of b1
 - The excess will simply be truncated, which could be confusing to programmers who write or use the code
 - This truncation is called object slicing



Design Issues for Object-Oriented Languages



- Second question here is concerned with those cases where objects are allocated from the heap
 - Question is whether deallocation is implicit, explicit, or both
 - If deallocation is implicit, some implicit method of storage reclamation is required
 - If deallocation can be explicit, that raises the issue of whether dangling pointers or references can be created

Dynamic and Static Binding

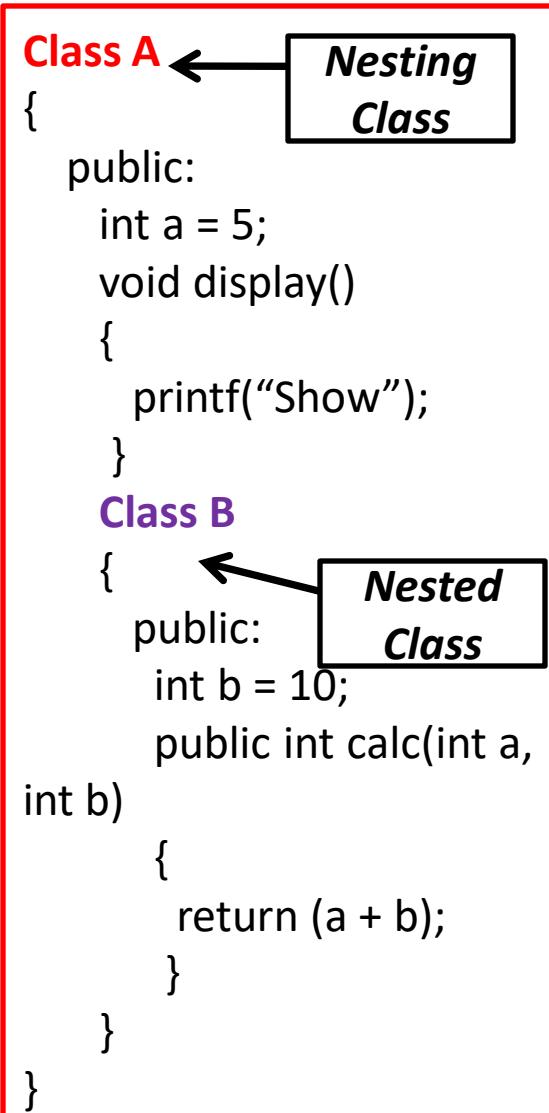
- Dynamic binding of messages to methods is an essential part of object-oriented programming
- Question here is whether all binding of messages to methods is dynamic
- Alternative is to allow the user to specify whether a specific binding is to be dynamic or static
- Advantage of this is that static bindings are faster
- So, if a binding need not be dynamic, why pay the price?

Design Issues for Object-Oriented Languages



Nested Classes

- One of the primary motivations for nesting class definitions is information hiding
- If a new class is needed by only one class, there is no reason to define it so it can be seen by other classes
 - New class can be nested inside the class that uses it
 - In some cases, the new class is nested inside a subprogram, rather than directly in another class
 - The class in which the new class is nested is called the nesting class
- Most obvious design issues associated with class nesting are related to visibility
- Specifically, one issue is: Which of the facilities of the nesting class are visible in the nested class
- The other main issue is the opposite: Which of the facilities of the nested class are visible in the nesting class?





Design Issues for Object-Oriented Languages

Initialization of Objects

- Initialization issue is whether and how objects are initialized to values when they are created
- First question is whether objects must be initialized manually or through some implicit mechanism
- When an object of a subclass is created, is the associated initialization of the inherited parent class member implicit or must the programmer explicitly deal with it



Support for Object-Oriented Programming in C++

- C++ was the first widely used object-oriented programming language and is still among the most popular
- So, naturally, it is the one with which other languages are often compared

General Characteristics

- To main backward compatibility with C, C++ retains the type system of C and adds classes to it
- C++ has both traditional imperative-language types and the class structure of an object-oriented language
- C++ is a hybrid language, supporting both procedural programming and object-oriented programming
- Objects of C++ can be static, stack dynamic or heap dynamic
- Explicit deallocation using the delete operator is required for heap-dynamic objects because C++ does not include implicit storage reclamation

Support for Object-Oriented Programming in C++



- Many class definitions include a destructor method, which is implicitly called when an object of the class ceases to exist
- Destructor is used to deallocate heap-allocated memory that is referenced by data members
 - It may also be used to record part or all of the state of the object just before it dies, usually for debugging purposes

Inheritance

- A C++ class can be derived from an existing class, which is then its parent, or base, class
- C++ class can also be stand-alone, without a superclass
- Data defined in a class definition -> Data Members of that class
- Functions defined in a class definition -> Member Functions of that class (Methods)
- Some or all of the members of the base class may be inherited by the derived class, which can also add new members and modify inherited member functions

Support for Object-Oriented Programming in C++



- All C++ objects must be initialized before they are used
- Therefore, all C++ classes include at least one constructor method that initializes the data members of the new object
- Constructor methods are implicitly called when an object is created
- If any of the data members are pointers to heap-allocated data, the constructor allocates that storage. Eg: int *a[] = new array[20];
- If a class has a parent, the inherited data members must be initialized when the subclass object is created
 - To do this, the parent constructor is implicitly called
- When initialization data must be furnished to the parent constructor, it is given in the call to the subclass object constructor

```
subclass (subclass parameters) : parent_class (superclass parameters) {  
    ...  
}
```



Miscellaneous

```
class Base
{
    int x;
public:
    Base(int i)
    {
        x = i;
        cout << "Base Parameterized Constructor\n";
        cout << i;
    }
};

class Derived : public Base
{
    int y;
public:
    Derived(int j, int p):Base(p)←
    {
        y = j;
        cout << "\nDerived Parameterized Constructor\n";
        cout << y;
    }
};
```

Receives 2 parameters as argument and passes one parameter to base class constructor

```
int main()
{
    Derived d(10, 5);
}
```

O/P: Base Parameterized Constructor
5
Derived Parameterized Constructor
10



Miscellaneous

```
class Base
{
    int x;
public:
    Base(int i)
    {
        x = i;
        cout << "Base Parameterized Constructor\n";
        cout << i;
    }
};

class Derived : public Base
{
    int y;
public:
    Derived(int j):Base(j)←
    {
        y = j;
        cout << "\nDerived Parameterized Constructor\n";
        cout << y;
    }
};
```

Receives 1 parameter as argument and passes that parameter to base class constructor

```
int main()
{
    Derived d(10);
}
```

O/P: Base Parameterized Constructor
10
Derived Parameterized Constructor
10



Support for Object-Oriented Programming

- If no constructor is included in a class definition, the compiler includes a trivial constructor
- This default constructor calls the constructor of the parent class, if there is a parent class

```
class derived_class_name : derivation_mode base_class_name  
{data member and member function declarations} ;
```

```
class base_class {  
private:  
    int a;  
    float x;  
protected:  
    int b;  
    float y;  
public:  
    int c;  
    float z;  
};
```

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

```
class subclass_1 : public base_class {...};  
class subclass_2 : private base_class {...};
```

- In subclass_1, b and y are protected, and c and z are public
- In subclass_2, b, y, c, and z are private
- No derived class of subclass_2 can have members with access to any member of base_class
- The data members a and x in base_class are not accessible in either subclass_1 or subclass_2



Support for Object-Oriented Programming

- Note that private-derived subclasses cannot be subtypes
- Eg: If the base class has a public data member, under private derivation that data member would be private in the subclass
- Therefore, if an object of the subclass were substituted for an object of the base class, accesses to that data member would be illegal on the subclass object
- The is-a relationship would be broken
- Under private class derivation, no member of the parent class is implicitly visible to the instances of the derived class
- Any member that must be made visible must be reexported in the derived class
- This reexportation in effect exempts a member from being hidden even though the derivation was private

```
class subclass_3 : private base_class {  
    base_class :: c;  
    ...  
}
```

Now, instances of subclass_3 can access c. As far as c is concerned, it is as if the derivation had been public. The double colon (::) in this class definition is a scope resolution operator. It specifies the class where its following entity is defined.



Miscellaneous

```
//Base class
class Parent
{
public:
    int id_p;
};

class Child : private Parent
{
public:
    int id_c;
};

class child_derived: public Child
{};

}
```

```
int main()
{
    Parent obj;
    obj.id_p = 9;
    Child obj1;
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is " << obj.id_p <<
endl;
    cout << "Parent id is " << obj1.id_p <<
endl;

    return 0;
}
```

Inheritance -> Is Unidirectional

Types

- Single Inheritance
- Multiple Inheritance

O/P: Error in line: obj1.id_p = 91;



Miscellaneous

```
#include <iostream>
using namespace std;
//Base class
class Parent
{
public:
    int id_p;
};

class Child : private Parent
{
public:
    int id_c;
    int Parent :: id_p;
};

class child_derived: public Child
{
```

```
int main()
{
    Parent obj;
    obj.id_p = 9;
    Child obj1;
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is " << obj.id_p <<
endl;
    cout << "Parent id is " << obj1.id_p <<
endl;

    return 0;
}
```

Inheritance -> Is Unidirectional

Types

- Single Inheritance
- Multiple Inheritance

O/P: Error in line: obj1.id_p = 91;



Support for Object-Oriented Programming

An example for the purpose and use of private derivation

```
class single_linked_list {  
private:  
    class node {  
        public:  
            node *link;  
            int contents;  
    };  
    node *head;  
public:  
    single_linked_list() {head = 0};  
    void insert_at_head(int);  
    void insert_at_tail(int);  
    int remove_at_head();  
    int empty();  
};
```

Nesting Class

Nested Class

A diagram showing the structure of the `single_linked_list` class. A dashed blue line encloses the `node` class definition, which is labeled **Nesting Class**. Another dashed blue line encloses the entire class definition, including the `node` class, and is labeled **Nested Class**. Arrows point from the labels to their respective dashed lines.

```
class stack : public single_linked_list {  
public:  
    stack() {}  
    void push(int value) {  
        insert_at_head(value);  
    }  
    int pop() {  
        return remove_at_head();  
    }  
};  
class queue : public single_linked_list {  
public:  
    queue() {}  
    void enqueue(int value) {  
        insert_at_tail(value);  
    }  
    int dequeue() {  
        remove_at_head();  
    }  
};
```

- Note that nested classes have no special access to members of the nesting class. Only static data members of the nesting class are visible to methods of the nested class
- Enclosing class, `single_linked_list`, has just a single data member, a pointer to act as the list's header. It contains a constructor function, which simply sets head to the null pointer value.



Support for Object-Oriented Programming

- A client of a stack object could call `insert_at_tail`, thereby destroying the integrity of its stack
- Likewise, a client of a queue object could call `insert_at_head`
- These unwanted accesses are allowed because both stack and queue are subtypes of `single_linked_list`
- Our two example derived classes can be written to make them not subtypes of their parent class by using private, rather than public, derivation
- Then, both will also need to reexport `empty`, because it will become hidden to their instances
- This situation illustrates the motivation for the private-derivation option



Support for Object-Oriented Programming

```
class stack_2 : private single_linked_list {
public:
    stack_2() {}
    void push(int value) {
        single_linked_list :: insert_at_head(value);
    }
    int pop() {
        return single_linked_list :: remove_at_head();
    }
    single_linked_list::empty();
};

class queue_2 : private single_linked_list {
public:
    queue_2() {}
    void enqueue(int value) {
        single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
        single_linked_list :: remove_at_head();
    }
    single_linked_list::empty();
};
```

Support for Object-Oriented Programming

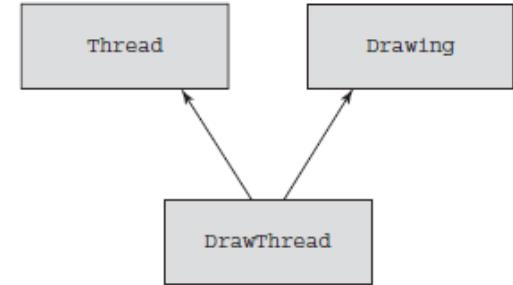
- The two versions of stack and queue illustrate the difference between subtypes and derived types that are not subtypes
- The linked list is a generalization of both stacks and queues, because both can be implemented as linked lists
- So, it is natural to inherit from a linked-list class to define stack and queue classes
- However, neither is a subtype of the linked-list class, because both make the public members of the parent class private, which makes them inaccessible to clients
- One of the reasons friends are necessary is that sometimes a subprogram must be written that can access the members of two different classes
- Eg: Suppose a program uses a class for vectors and one for matrices, and a subprogram is needed to multiply a vector object times a matrix object
- In C++, the multiply function can be made a friend of both classes

Support for Object-Oriented Programming



- C++ provides multiple inheritance, which allows more than one class to be named as the parent of a new class

```
class Thread { ... };
class Drawing { ... };
class DrawThread : public Thread, public Drawing { ... };
```



- Class DrawThread inherits all of the members of both Thread and Drawing
- If both Thread and Drawing happen to include members with the same name, they can be unambiguously referenced in objects of class DrawThread by using the scope resolution operator (::)
- Overriding methods in C++ must have exactly the same parameter profile as the overridden method
- If there is any difference in the parameter profiles, the method in the subclass is considered a new method that is unrelated to the method with the same name in the ancestor class
- The return type of the overriding method either must be the same as that of the overridden method or must be a publicly derived type of the return type of the overridden method



Miscellaneous

```
class A
{
    int idA;
    void setId(int i) ←
    {
        idA = i;
    }
    int getId()
    {
        return idA;
    }
};
```

```
class B
{
    int idB;
    void setId(int i) ←
    {
        idB = i;
    }
    int getId()
    {
        return idB;
    }
};
class AB : public A, public B
{
};
int main()
{
    AB *ab = new AB();
    ab -> setId(5);
}
```

**Problem in
Multiple
Inheritance**

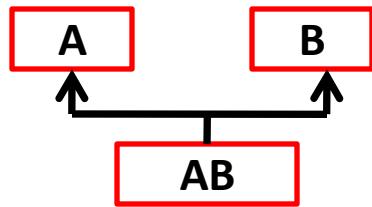
O/P: Error: request for member 'setId' is ambiguous
Note: candidates are: **void B::setId(int)**
Note: **void A::setId(int)**

Miscellaneous

```
class A
{
public:
    int idA;
void setId(int i)
{
    idA = i;
}
int getId()
{
    return idA;
}
};
```

Note: Either use the one defined in “main” function (::) or use the one defined in class AB (using)

```
class B
{
public:
    int idB;
void setId(int i)
{
    idB = i;
}
int getId()
{
    return idB;
};
class AB : public A, public B
{
public:
using A::setId;
using A::getId;
};
int main()
{
    AB *ab = new AB();
    ab->B::setId(10);
```



Miscellaneous

```

class base
{
public:
    virtual void print ()
    {
        cout<< "print base class" << endl;
    }

    void show ()
    {
        cout<< "show base class" << endl;
    }
};

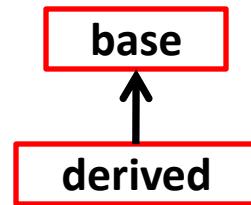
class derived: public base
{
public:
    void print () //print () is already virtual
    function in derived class, we could also
    declared as virtual void print () explicitly
    {
        cout<< "print derived class" << endl;
    }

    void show ()
    {
        cout<< "show derived class" << endl;
    }
};

O/P: print derived class
        show base class
  
```

```

int main()
{
    base *bptr;
    derived d;
    bptr = &d;
}
  
```



//virtual function, binded at runtime (Runtime polymorphism)
`bptr->print();`

// Non-virtual function, binded at compile time
`bptr->show();`
`return 0;`

This type of polymorphism is achieved by Function Overriding. Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden

Types

- Functional Overloading (Compile Time)
- Operator Overloading (Compile Time)
- **Virtual Functions (Run Time)**



Dynamic Binding

- All of the member functions we have defined thus far are statically bound; that is, a call to one of them is statically bound to a function definition
- A C++ object could be manipulated through a value variable, rather than a pointer or a reference (Such an object would be static or stack dynamic)
- However, in that case, the object's type is known and static, so dynamic binding is not needed
- On the other hand, a pointer variable that has the type of a base class can be used to point to any heap-dynamic objects of any class publicly derived from that base class, making it a polymorphic variable
- Publicly derived subclasses are subtypes if none of the members of the base class are private
- Privately derived subclasses are never subtypes
- A pointer to a base class cannot be used to reference a method in a subclass that is not a subtype



Miscellaneous

```
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
    virtual void draw()
    {
        cout<<"Vehicle Draw";
    }
};
```

FourWheeler *obj = new Vehicle();
O/P: error: invalid conversion from
'Vehicle*' to 'FourWheeler*' [-fpermissive]

```
class FourWheeler: public Vehicle
{
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler
Vehicle" << endl;
    }
    void draw()
    {
        cout<<"Fourwheeler Draw";
    }
};

void main()
{
    Vehicle *obj = new FourWheeler();
    obj->draw();
}
```

O/P: This is a Vehicle
This is a 4 wheeler Vehicle
Fourwheeler Draw



Dynamic Binding

- C++ does not allow value variables (as opposed to pointers or reference) to be polymorphic
- When a polymorphic variable is used to call a member function overridden in one of the derived classes, the call must be dynamically bound to the correct member function definition
- Member functions that must be dynamically bound must be declared to be virtual functions by preceding their headers with the reserved word `virtual`, which can appear only in a class body

```
class Shape {  
public:  
    virtual void draw() = 0;  
    ...  
};  
class Circle : public Shape {  
public:  
    void draw() { ... }  
    ...  
};  
class Rectangle : public Shape {  
public:  
    void draw() { ... }  
    ...  
};  
class Square : public Rectangle {  
public:  
    void draw() { ... }  
    ...  
};
```

```
Square* sq = new Square;  
Rectangle* rect = new Rectangle;  
Shape* ptr_shape;  
  
ptr shape = sq; // Now ptr_shape points to a  
// Square object  
ptr_shape->draw(); // Dynamically bound to the draw  
// in the Square class  
rect->draw(); // Statically bound to the draw  
// in the Rectangle class
```

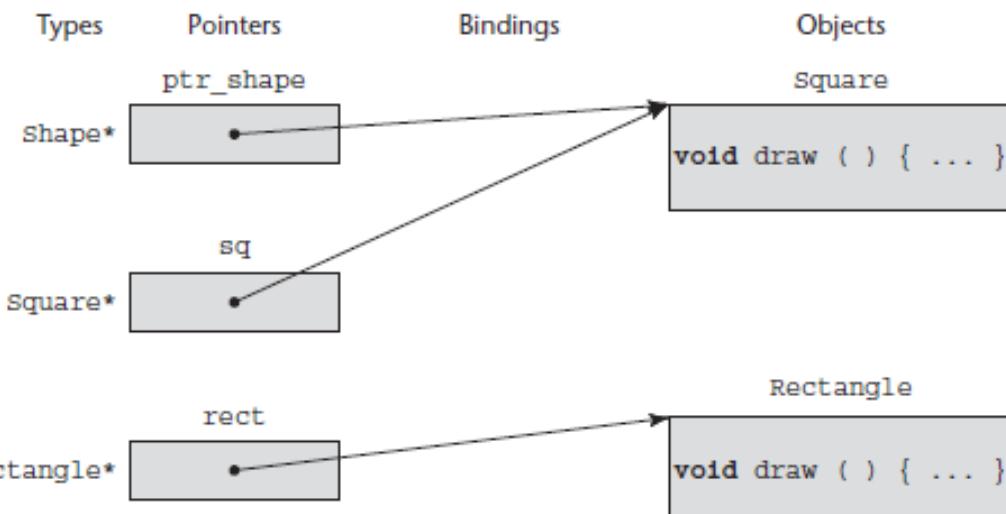
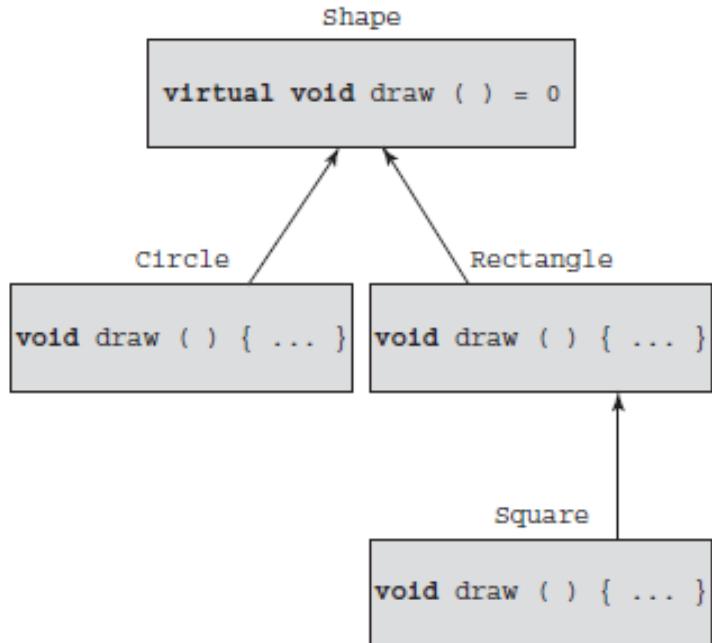
- `Rect` -> Value Variable
- `Ptr_shape` -> Polymorphic Variable

Versions of `draw` must be defined to be virtual. When a call to `draw` is made with a pointer to the base class of the derived classes, that call must be dynamically bound to the member function of the correct derived class.



Dynamic Binding

Class Hierarchy





Dynamic Binding

- Notice that the draw function in the definition of the base class shape is set to 0
- This peculiar syntax is used to indicate that this member function is a pure virtual function, meaning that it has no body and it cannot be called
- It must be redefined in derived classes if they call the function
- The purpose of a pure virtual function is to provide the interface of a function without giving any of its implementation
- Pure virtual functions are usually defined when an actual member function in the base class would not be useful
- Any class that includes a pure virtual function is an abstract class
- An abstract class can include completely defined methods
- It is illegal to instantiate an abstract class
- If a subclass of an abstract class does not redefine a pure virtual function of its parent class, that function remains as a pure virtual function in the subclass and the subclass is also an abstract class
- Abstract classes and inheritance together support a powerful technique for software development



Dynamic Binding

- Dynamic binding allows the code that uses members like draw to be written before all or even any of the versions of draw are written
- New derived classes could be added years later, without requiring any change to the code that uses such dynamically bound members
- This is a highly useful feature of object-oriented languages
- Reference assignments for stack-dynamic objects are different from pointer assignments for heap-dynamic objects

```
Square sq;           // Allocate a Square object on the stack
Rectangle rect;    // Allocate a Rectangle object on
                   // the stack
rect = sq;          // Copies the data member values from
                   // the Square object
rect.draw();        // Calls the draw from the Rectangle
                   // object
```

- In the assignment rect = sq, the member data from the object referenced by sq would be assigned to the data members of the object referenced by rect, but rect would still reference the Rectangle object
- Therefore, the call to draw through the object referenced by rect would be that of the Rectangle class
- If rect and sq were pointers to heap-dynamic objects, the same assignment would be a pointer assignment, which would make rect point to the Square object, and a call to draw through rect would be bound dynamically to the draw in the Square object

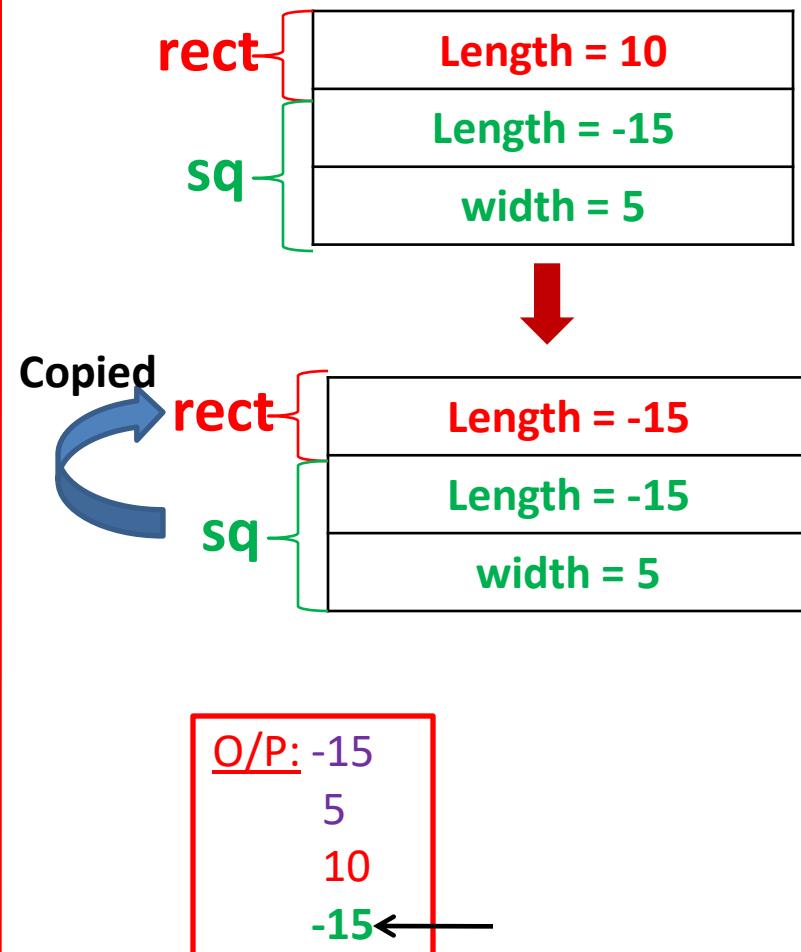
Miscellaneous

```

class Rectangle
{
    int Length;
    void draw() { .... }
};

class Square : public Rectangle
{
    int width;
    void draw() { .... }
};

int main()
{
    Square sq;
    sq.Length = -15;
    cout << sq.Length << endl;
    sq.width = 5;
    cout << sq.width << endl;
    Rectangle rect;
    rect.Length = 10;
    cout << rect.Length << endl;
    rect = sq;
    cout << rect.Length << endl;
    rect.draw() ←
}
  
```



It will call the method “draw” in class “**Rectangle**”

Note: In the assignment `rect = sq`, the member data from the object referenced by `sq` would be assigned to the data members of the object referenced by `rect`, but `rect` would still reference the `Rectangle` object

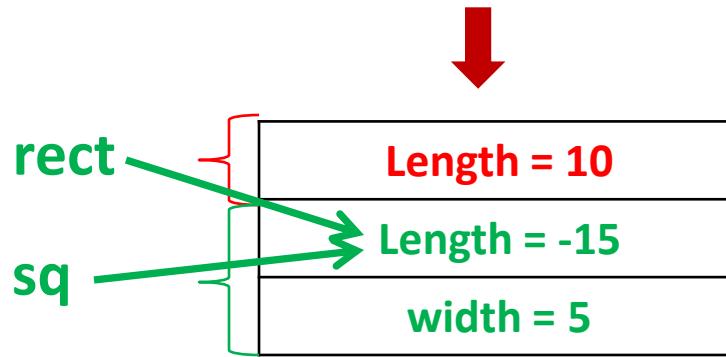
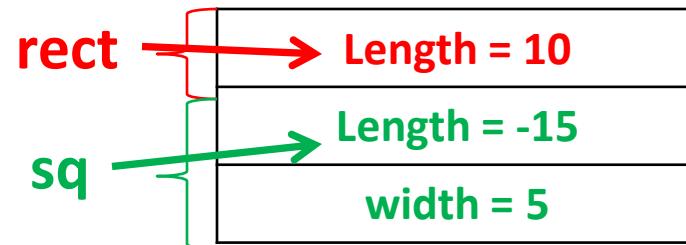
Miscellaneous

```

class Rectangle
{
    int Length;
    void draw() { .... }
};

class Square : public Rectangle
{
    int width;
    void draw() { .... }
};

int main()
{
    Square *sq;
    sq.Length = -15;
    sq.width = 5;
    Rectangle *rect;
    rect.Length = 10;
    rect = sq;
    rect.draw()
}
  
```



Note: If `rect` and `sq` were pointers to heap-dynamic objects, the same assignment would be a pointer assignment, which would make `rect` point to the `Square` object, and a call to `draw` through `rect` would be bound dynamically to the `draw` in the `Square` object.

It will call the method “draw” in class “**Square**”



Implementation of Object-Oriented Constructs

- Two parts of language support for object-oriented programming that pose interesting questions for language implementers -> Storage structures for instance variables; Dynamic bindings of messages to methods

Instance Data Storage

- In C++, classes are defined as extensions of C's record structures—structs
- Similarity suggests a storage structure for the instance variables of class instances—that of a record -> Class Instance Record (CIR)
- Structure of a CIR is static, so it is built at compile time and used as a template for the creation of the data of class instances
- Every class has its own CIR
- When a derivation takes place, the CIR for the subclass is a copy of that of the parent class, with entries for the new instance variables added at the end
- Because the structure of the CIR is static, access to all instance variables can be done as it is in records, using constant offsets from the beginning of the CIR instance
- This makes these accesses as efficient as those for the fields of records

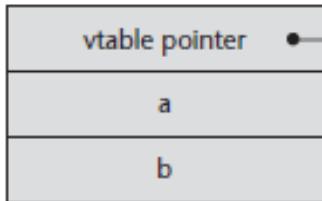
Implementation of Object-Oriented Constructs

Java:

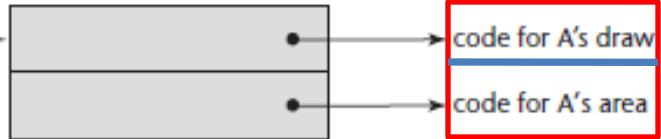
```
public class A {
    public int a, b;
    public void draw() { ... }
    public int area() { ... }
}
```

```
public class B extends A {
    public int c, d;
    public void draw() { ... }
    public void sift() { ... }
}
```

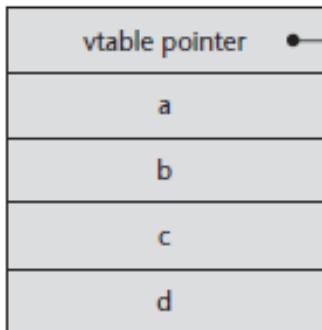
Class instance Record for A



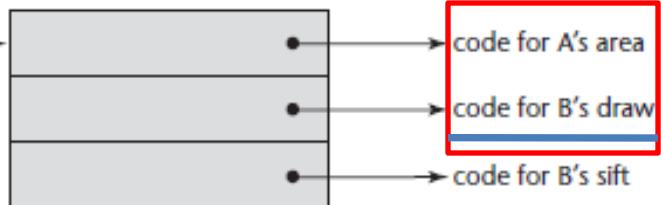
vtable for A



Class instance Record for B



vtable for B





Dynamic Binding of Method Calls to Methods

- Methods in a class that are statically bound need not be involved in the CIR for the class
- However, methods that will be dynamically bound must have entries in this structure
- Such entries could simply have a pointer to the code of the method, which must be set at object creation time
- Calls to a method could then be connected to the corresponding code through this pointer in the CIR
- Drawback to this technique is that every instance would need to store pointers to all dynamically bound methods that could be called from the instance



Dynamic Binding of Method Calls to Methods

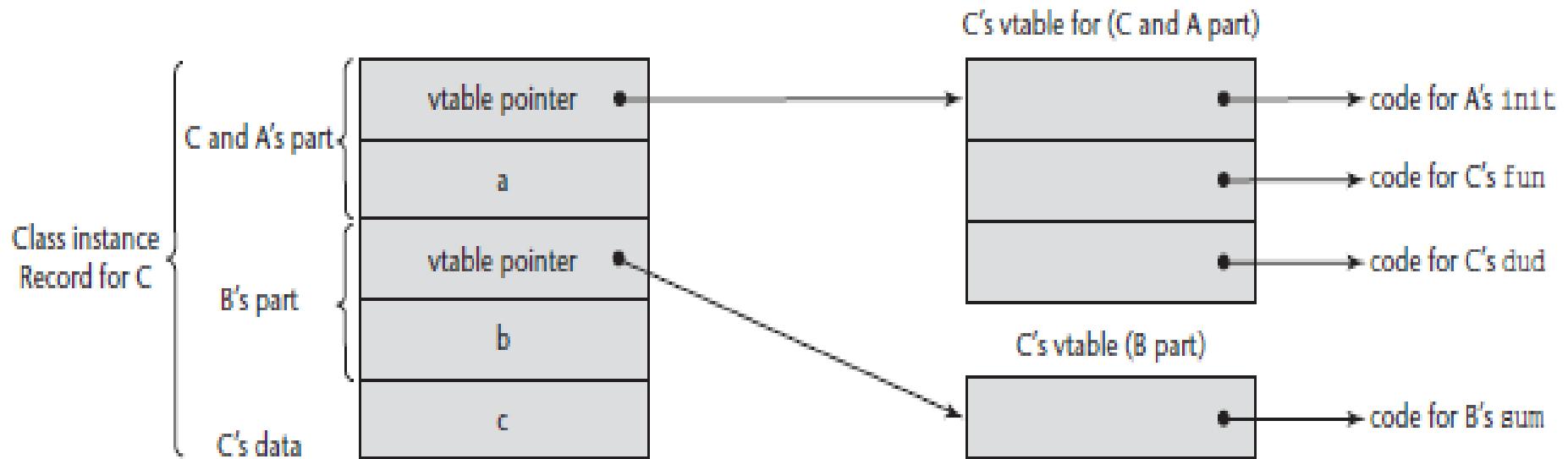
- Notice that the list of dynamically bound methods that can be called from an instance of a class is the same for all instances of that class
- Therefore, the list of such methods must be stored only once
- So the CIR for an instance needs only a single pointer to that list to enable it to find called methods
- The storage structure for the list is often called a virtual method table (vtable)
- Method calls can be represented as offsets from the beginning of the vtable
- Polymorphic variables of an ancestor class always reference the CIR of the correct type object, so getting to the correct version of a dynamically bound method is assured
- Multiple inheritance complicates the implementation of dynamic binding

Dynamic Binding of Method Calls to Methods



```
class A {  
public:  
    int a;  
    virtual void fun() { ... }  
    virtual void init() { ... }  
};  
class B {
```

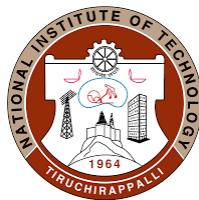
```
public:  
    int b;  
    virtual void sum() { ... }  
};  
class C : public A, public B {  
public:  
    int c;  
    virtual void fun() { ... }  
    virtual void dud() { ... }  
};
```



Dynamic Binding of Method Calls to Methods



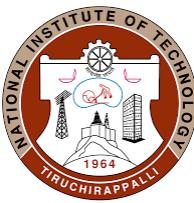
- C class inherits the variable a and the init method from the A class
- It redefines the fun method, although both its fun and that of the parent class A are potentially visible through a polymorphic variable (of type A)
- From B, C inherits the variable b and the sum method
- C defines its own variable, c, and defines an uninherited method, dud
- A CIR for C must include A's data, B's data, and C's data, as well as some means of accessing all visible methods
- Under single inheritance, the CIR would include a pointer to a vtable that has the addresses of the code of all visible methods
- With multiple inheritance, however, it is not that simple
- There must be at least two different views available in the CIR—one for each of the parent classes, one of which includes the view for the subclass, C
- This inclusion of the view of the subclass in the parent class's view is just as in the implementation of single inheritance
- There must also be two vtables: one for the A and C view and one for the B view
- The first part of the CIR for C in this case can be the C and A view, which begins with a vtable pointer for the methods of C and those inherited from A, and includes the data inherited from A
- Following this in C's CIR is the B view part, which begins with a vtable pointer for the virtual methods of B, which is followed by the data inherited from B and the data defined in C



CSPC31: Principles of Programming Languages

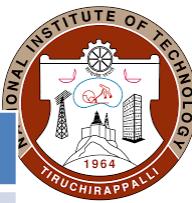
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 13 – Concurrency

Introduction to Subprogram-Level Concurrency



- A task is a unit of a program, similar to a subprogram, that can be in concurrent execution with other units of the same program
- Each task in a program can support one thread of control
- ~~Tasks are sometimes called processes~~
- ~~In some languages, for example Java and C#, certain methods serve as tasks. Such methods are executed in objects called threads~~
- Tasks fall into two general categories: heavyweight and lightweight
 - A heavyweight task executes in its own address space
 - Lightweight tasks all run in the same address space
- It is easier to implement lightweight tasks than heavyweight tasks
- Furthermore, lightweight tasks can be more efficient than heavyweight tasks, because less effort is required to manage their execution
- A task can communicate with other tasks through shared nonlocal variables, through message passing, or through parameters
- If a task does not communicate with or affect the execution of any other task in the program in any way, it is said to be disjoint
- Because tasks often work together to create simulations or solve problems and therefore are not disjoint, they must use some form of communication to either synchronize their executions or share data or both



Introduction to Subprogram-Level Concurrency

- Synchronization is a mechanism that controls the order in which tasks execute
- Two kinds of synchronization are required when tasks share data: cooperation and competition
- Cooperation synchronization is required between task A and task B when task A must wait for task B to complete some specific activity before task A can begin or continue its execution -> **Producer-Consumer Problem**
- Competition synchronization is required between two tasks when both require the use of some resource that cannot be simultaneously used -> **Bank Transaction**



Introduction to Subprogram-Level Concurrency

- Cooperation Synchronization (Producer-Consumer Problem)
 - The consumer unit must not be allowed to take data from the buffer if the buffer is empty
 - Likewise, the producer unit cannot be allowed to place new data in the buffer if the buffer is full
 - This is a problem of cooperation synchronization because the users of the shared data structure must cooperate if the buffer is to be used correctly
- Competition synchronization (Bank Transaction)
 - Prevents two tasks from accessing a shared data structure at exactly the same time—a situation that could destroy the integrity of that shared data
 - To provide competition synchronization, mutually exclusive access to the shared data must be guaranteed



Cooperation Synchronization

- This problem originated in the development of operating systems, in which one program unit produces some data value or resource and another uses it
- Produced data are usually placed in a storage buffer by the producing unit and removed from that buffer by the consuming unit
- The sequence of stores to and removals from the buffer must be synchronized
- The consumer unit must not be allowed to take data from the buffer if the buffer is empty
- Likewise, the producer unit cannot be allowed to place new data in the buffer if the buffer is full
- This is a problem of cooperation synchronization because the users of the shared data structure must cooperate if the buffer is to be used correctly



Competition Synchronization

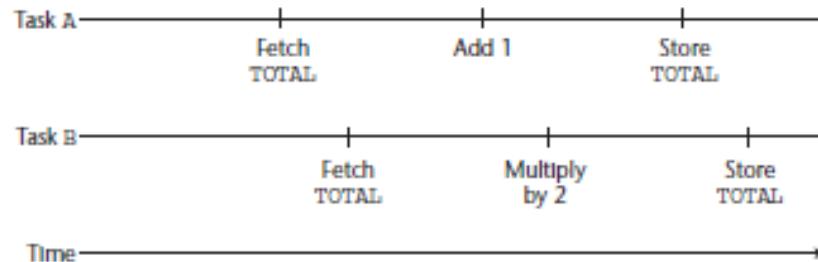
To clarify the competition problem, consider the following scenario: Suppose task A has the statement `TOTAL += 1`, where `TOTAL` is a shared integer variable. Furthermore, suppose task B has the statement `TOTAL *= 2`. Task A and task B could try to change `TOTAL` at the same time.

At the machine language level, each task may accomplish its operation on `TOTAL` with the following three-step process:

1. Fetch the value of `TOTAL`.
2. Perform the arithmetic operation.
3. Put the new value back in `TOTAL`.

Without competition synchronization, given the previously described operations performed by tasks A and B on `TOTAL`, four different values could result, depending on the order of the steps of the operation. Assume `TOTAL` has the value 3 before either A or B attempts to modify it. If task A completes its operation before task B begins, the value will be 8, which is assumed here to be correct. But if both A and B fetch the value of `TOTAL` before either task puts its new value back, the result will be incorrect. If A puts its value back first, the value of `TOTAL` will be 6. This case is shown in Figure 13.1. If B puts its value back first, the value of `TOTAL` will be 4. Finally, if B completes its operation before task A begins, the value will be 7. A situation that leads to these problems is sometimes called a **race condition**, because two or more tasks are racing to use the shared resource and the behavior of the program depends on which task arrives first (and wins the race). The importance of competition synchronization should now be clear.

Value of `TOTAL`: 3 —————— 4 6





Solution

- Three methods of providing for mutually exclusive access to a shared resource are semaphores, monitors, and message passing
- Semaphores:
- Use a variable called Semaphore variable, X
Wait(P)↓; Signal(V) ↑
- The wait semaphore subprogram is used to test the counter of a given semaphore variable
 - If the value is greater than zero, the caller can carry out its operation. In this case, the counter value of the semaphore variable is decremented to indicate that there is now one fewer of whatever it counts
 - If the value of the counter is zero, the caller must be placed on the waiting queue of the semaphore variable, and the processor must be given to some other ready task
- The release semaphore subprogram is used by a task to allow some other task to have one of whatever the counter of the specified semaphore variable counts
 - If the queue of the specified semaphore variable is empty, which means no task is waiting, release increments its counter (to indicate there is one more of whatever is being controlled that is now available)
 - If one or more tasks are waiting, release moves one of them from the semaphore queue to the ready queue

P(s)
Critical Section
V(s)



Solution

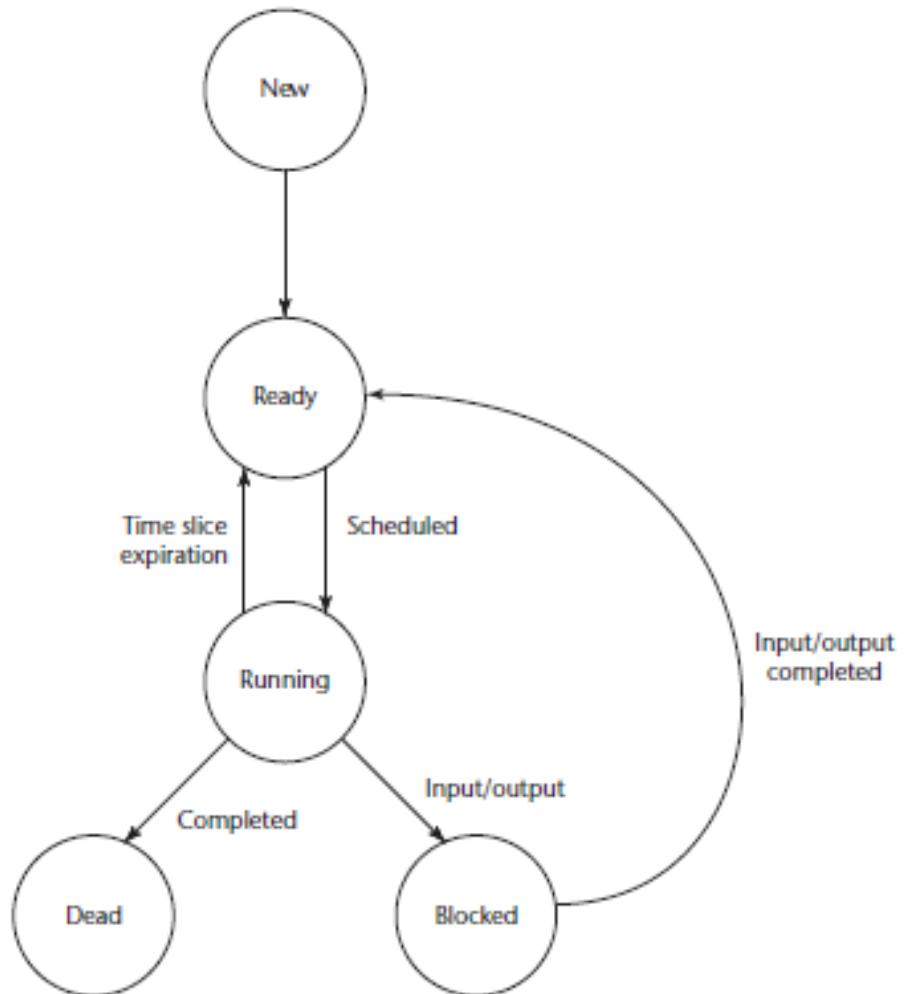
- Semaphore Types
 - Binary Semaphore (Solution for Competition synchronization – Provides mutually exclusive access to shared resource)
 - If the value of the semaphore variable is 0, then wait
 - Else
 - Before entering the critical section, make the semaphore value as 0
 - Execute the code in critical section
 - After executing the code in critical section, make the semaphore value as 1 and then exit from critical section
 - Counting Semaphore (Solution for Competition Semaphore)
 - Check the value of the semaphore variable value
 - **Producer:** If the value has not reached the maximum count
 - Increment the semaphore variable value by 1
 - Manufacture a product
 - Else, wait and repeat the checking after sometime
 - **Consumer:** If the value is not 0
 - Decrement the semaphore variable value by 1
 - Consume the product
 - Else, wait and repeat the checking after sometime



Various States of Task

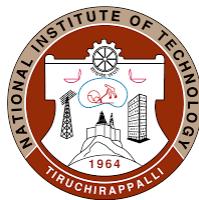
- Tasks can be in several different states:
1. **New:** A task is in the new state when it has been created but has not yet begun its execution
 2. **Ready:** A ready task is ready to run but is not currently running. Either it has not been given processor time by the scheduler, or it had run previously but was blocked. Tasks that are ready to run are stored in a queue that is often called the task ready queue
 3. **Running:** A running task is one that is currently executing; that is, it has a processor and its code is being executed
 4. **Blocked:** A task that is blocked has been running, but that execution was interrupted by one of several different events, the most common of which is an input or output operation. In addition to input and output, some languages provide operations for the user program to specify that a task is to be blocked
 5. **Dead:** A dead task is no longer active in any sense. A task dies when its execution is completed or it is explicitly killed by the program

Various States of Task





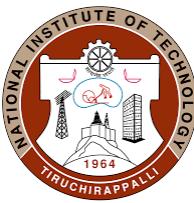
Thank You



CSPC31: Principles of Programming Languages

Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 15 – Functional Programming Languages

Introduction



- Imperative languages are designed for von Neumann architecture
- Functional programming paradigm, which is based on mathematical functions, is the design basis of the most important nonimperative styles of languages
- Purely functional programs are easier to understand, both during and after development, largely because the meanings of expressions are independent of their context (one characterizing feature of a pure functional programming language is that neither expressions nor functions have side effects)
- One of the fundamental characteristics of programs written in imperative languages is that they have state, which changes throughout the execution process
 - This state is represented by the program's variables
 - The author and all readers of the program must understand the uses of its variables and how the program's state changes through execution
 - For a large program, this is a daunting task
- This is one problem with programs written in an imperative language that is not present in a program written in a pure functional language, for such programs have neither variables nor state



Introduction

- LISP began as a pure functional language but soon acquired some important imperative features that increased its execution efficiency
 - It is still the most important of the functional languages, at least in the sense that it is the only one that has achieved widespread use
 - It dominates in the areas of knowledge representation, machine learning, intelligent training systems, and the modeling of speech
- Scheme is a small, static-scoped dialect of LISP
- Scheme has been widely used to teach functional programming
- It is also used in some universities to teach introductory programming courses
- Functional programming languages are now being used in areas such as database processing, financial modeling, statistical analysis, and bio-informatics



Mathematical Functions

- A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set
- A function definition specifies the domain and range sets, either explicitly or implicitly, along with the mapping
- The mapping is described by an expression or, in some cases, by a table
- Functions are often applied to a particular element of the domain set, given as a parameter to the function
- Note that the domain set may be the cross product of several sets (reflecting that there can be more than one parameter)
- A function yields an element of the range set

Fundamental Characteristics of Mathematical Functions



Sl. No.	Mathematical Functions	Imperative Programming Languages
1.	Evaluation order of their mapping expressions is controlled by recursion and conditional expressions	Evaluation order is controlled by the sequencing and iterative repetition
2.	As they have no side effects and cannot depend on any external values, they always map a particular element of the domain to the same element of the range	A subprogram may depend on the current values of several nonlocal or global variables. This makes it difficult to determine statically what values the subprogram will produce and what side effects it will have on a particular execution
3.	In mathematics, there is no such thing as a variable that models a memory location. Hence, there is no concept of the state of a function. A mathematical function maps its parameter(s) to a value (or values), rather than specifying a sequence of operations on values in memory to produce a value	Local variables in functions maintain the state of the function. Computation is accomplished by evaluating expressions in assignment statements that change the state of the program



Simple Functions

- Function definitions are often written as a function name, followed by a list of parameters in parentheses, followed by the mapping expression
- For example, **cube(x) = x * x * x**, where **x is a real number**
- In this definition, the **domain and range sets are the real numbers**
- The symbol = is used to mean “is defined as”
- The parameter x can represent any member of the domain set, but it is fixed to represent one specific element during evaluation of the function expression
- This is one way the parameters of mathematical functions differ from the variables in imperative languages
- Function applications are specified by pairing the function name with a particular element of the domain set
- The range element is obtained by evaluating the function-mapping expression with the domain element substituted for the occurrences of the parameter
- Once again, it is important to note that during evaluation, the mapping of a function contains no unbound parameters, where a bound parameter is a name for a particular value



Simple Functions

- Every occurrence of a parameter is bound to a value from the domain set and is a constant during evaluation
- For example, consider the following evaluation of $\text{cube}(x)$:

$$\text{cube}(\mathbf{2.0}) = 2.0 * 2.0 * 2.0 = 8$$

- The parameter x is bound to 2.0 during the evaluation and there are no unbound parameters
- Furthermore, x is a constant (its value cannot be changed) during the evaluation
- Early theoretical work on functions separated the task of defining a function from that of naming the function
- Lambda notation provides a method for defining nameless functions
- A lambda expression specifies the parameters and the mapping of a function
- **The lambda expression is the function itself, which is nameless**
- For example, consider the following lambda expression:

$$\lambda(x)x * x * x$$



Simple Functions

- As stated earlier, before evaluation a parameter represents any member of the domain set, but during evaluation it is bound to a particular member
- When a lambda expression is evaluated for a given parameter, the expression is said to be applied to that parameter
- The mechanics of such an application are the same as for any function evaluation
- Application of the example lambda expression is denoted as in the following example:

$$\lambda((x)x * x * x)(2)$$

which results in the value 8

- Lambda expressions, like other function definitions, can have more than one parameter



Functional Forms

- A higher-order function, or functional form, is one that either takes one or more functions as parameters or yields a function as its result, or both
- One common kind of functional form is function composition, which has two functional parameters and yields a function whose value is the first actual parameter function applied to the result of the second
- Function composition is written as an expression, using \circ as an operator, as in

$$b \equiv f \circ g$$

For example, if

$$f(x) \equiv x + 2$$

$$g(x) \equiv 3 * x$$

then b is defined as

$$b(x) \equiv f(g(x)), \text{ or } b(x) \equiv (3 * x) + 2$$



Functional Forms

- Apply-to-all is a functional form that takes a single function as a parameter
- If applied to a list of arguments, apply-to-all applies its functional parameter to each of the values in the list argument and collects the results in a list or sequence
- Apply-to-all is denoted by α

Let

$$b(x) \equiv x * x$$

then

$$\alpha(b, (2, 3, 4)) \text{ yields } (4, 9, 16)$$



Fundamentals of Functional Programming Languages

- Objective of the design of a functional programming language is to mimic mathematical functions to the greatest extent possible
- This results in an approach to problem solving that is fundamentally different from approaches used with imperative languages
- In an imperative language, an expression is evaluated and the result is stored in a memory location, which is represented as a variable in a program
- This is the purpose of assignment statements

$$(x + y) / (a - b)$$

- A purely functional programming language does not use variables or assignment statements, thus freeing the programmer from concerns related to the memory cells, or state, of the program
- Without variables, iterative constructs are not possible, for they are controlled by variables
- Repetition must be specified with recursion rather than with iteration
- Without variables, the execution of a purely functional program has no state in the sense of operational and denotational semantics
- The execution of a function always produces the same result when given the same parameters -> Referential Transparency



First Functional Programming

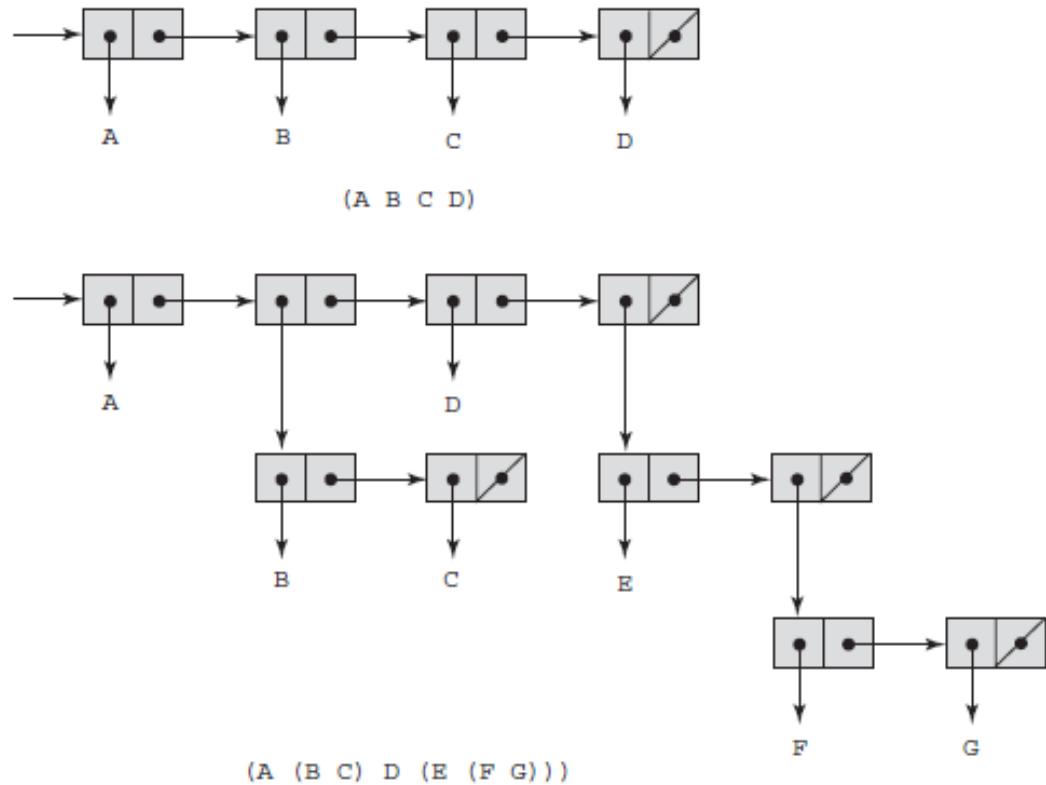
Language: LISP

- Lists are specified in LISP by delimiting their elements with parentheses
- The elements of simple lists are restricted to atoms, as in: (A B C D)
- Nested list structures are also specified by parentheses. For example, the list: (A (B C) D (E (F G)))

The first is the atom A; the second is the sublist (B C); the third is the atom D; the fourth is the sublist (E (F G)), which has as its second element the sublist (F G).

Internally, a list is usually stored as linked list structure in which each node has two pointers, one to reference the data of the node and the other to form the linked list. A list is referenced by a pointer to its first element.

The last element of a list has no successor, so its link is nil





Introduction

- Function calls were specified in a prefix list form originally called Cambridge Polish, as in the following:

(function_name argument₁ argument_n)

- For example, if **+** is a function that takes two or more numeric parameters, then the following two expressions evaluate to 12 and 20, respectively:

(+ 5 7)

(+ 3 4 7 6)

(function_name (LAMBDA (arg1 ... argn) expression))

- Nameless functions are sometimes useful in functional programming
- For example, consider a function whose action is to produce a function for immediate application to a parameter list
- The produced function has no need for a name, for it is applied only at the point of its construction
- Another feature of early LISP systems that was apparently accidental was the use of dynamic scoping
- Functions were evaluated in the environments of their callers
- No one at the time knew much about scoping, and there may have been little thought given to the choice
- Dynamic scoping was used for most dialects of LISP before 1975



Introduction to Scheme

- It is characterized by its small size, its exclusive use of static scoping, and its treatment of functions as first-class entities
- As first-class entities, Scheme functions can be the values of expressions, elements of lists, passed as parameters, and returned from functions
- Early versions of LISP did not provide all of these capabilities
- Scheme includes primitive functions for the basic arithmetic operations
- These are +, -, *, and /, for add, subtract, multiply, and divide
- * and + can have zero or more parameters
- If * is given no parameters, it returns 1
- if + is given no parameters, it returns 0
- + adds all of its parameters together
- * multiplies all its parameters together
- / and – can have two or more parameters
- In the case of subtraction, all but the first parameter are subtracted from the first
- Division is similar to subtraction

Expression	Value
42	42
(* 3 7)	21
(+ 5 7 8)	20
(- 5 6)	-1
(- 15 7 2)	6
(- 24 (* 4 3))	12

```
(display (* 3 7))  
Output: 21
```



Introduction to Scheme

- There are a large number of other numeric functions in Scheme, among them MODULO, ROUND, MAX, MIN, LOG, SIN, and SQRT
- SQRT returns the square root of its numeric parameter, if the parameter's value is not negative
- If the parameter is negative, SQRT yields a complex number
- In Scheme, note that we use uppercase letters for all reserved words and predefined functions
- The official definition of the language specifies that there is no distinction between uppercase and lowercase in these
- However, some implementations, for example DrRacket's teaching languages, require lowercase for reserved words and predefined functions
- If a function has a fixed number of parameters, such as SQRT, the number of parameters in the call must match that number
- If not, the interpreter will produce an error message



Defining Functions

- A Scheme program is a collection of function definitions
- Consequently, knowing how to define these functions is a prerequisite to writing the simplest program
- In Scheme, a nameless function actually includes the word LAMBDA, and is called a lambda expression

(LAMBDA (x) (* x x))

is a nameless function that returns the square of its given numeric parameter

- This function can be applied in the same way that named functions are: by placing it in the beginning of a list that contains the actual parameters
- For example, the following expression yields 49:
$$((\text{LAMBDA } (\text{x}) (\text{* } \text{x} \text{ x})) \text{ 7})$$

(display ((lambda (x) (* x x)) 7))

Output: 49
- In this expression, x is called a bound variable within the lambda expression
- A bound variable never changes in the expression after being bound to an actual parameter value at the time evaluation of the lambda expression begins
- Lambda expressions can have any number of parameters
- For example, we could have the following:

(LAMBDA (a b c x) (+ (* a x x) (* b x) c))



Introduction to Scheme

- Scheme special form function DEFINE serves two fundamental needs of Scheme programming:
 - To bind a name to a value
 - To bind a name to a lambda expression
- The form of DEFINE that binds a name to a value may make it appear that DEFINE can be used to create imperative language – style variables
- However, these name bindings create named values, not variables

```
(define pi 3.14159)  
(define two_pi (* 2 pi))  
(display pi)  
(newline)  
(display two_pi)
```

Output:

```
3.14159  
6.28318
```

- DEFINE is analogous to a declaration of a named constant in an imperative language

```
final float PI = 3.14159;  
final float TWO_PI = 2.0 * PI;
```



Introduction to Scheme

- Second use of the DEFINE function is to bind a lambda expression to a name
- In this case, the lambda expression is abbreviated by removing the word LAMBDA
- To bind a name to a lambda expression, DEFINE takes two lists as parameters
- The first parameter is the prototype of a function call, with the function name followed by the formal parameters, together in a list
- The second list contains an expression to which the name is to be bound
- The general form of such a DEFINE is:

(**DEFINE** (**function_name** **parameters**)
(expression)
)

```
(define (square number) (* number number))  
(display (square 5))
```

Output: 25



Introduction to Scheme

```
(define (sum a b) (display "x + y = ") (display (+ a b)))  
(sum 10 25)  
(newline)
```

(or)

```
(define (sum a b)  
(display "x + y = ")  
(display (+ a b)))  
(sum 10 25)  
(newline)
```

Output:
x + y = 35



Introduction to Scheme

- To illustrate the difference between primitive functions and the DEFINE special form, consider the following:

(DEFINE x 10)

- If DEFINE were a primitive function, EVAL's first action on this expression would be to evaluate the two parameters of DEFINE
- If x were not already bound to a value, this would be an error
- Furthermore, if x were already defined, it would also be an error, because this DEFINE would attempt to redefine x, which is illegal
- Remember, x is the name of a value; it is not a variable in the imperative sense
- Following is another example of a function
- It computes the length of the hypotenuse (the longest side) of a right triangle, given the lengths of the two other sides

(DEFINE (**hypotenuse** side1 side2))

(**SQRT**(+(**square** side1)(**square** side2)))

(define (**square** number) (* number number))

)

- Notice that hypotenuse uses square, which was defined previously



Output Functions

- Scheme includes a formatted output function, PRINTF, which is similar to the printf function of C
- Numeric Predicate Functions:
 - A predicate function is one that returns a Boolean value (some representation of either true or false)
 - Scheme includes a collection of predicate functions for numeric data

Notice that the names for all predefined predicate functions that have words for names end with question marks. In Scheme, the two Boolean values are #T and #F (or #t and #f), although some implementations use the empty list for false.⁵ The Scheme predefined predicate functions return the empty list, (), for false. When a list is interpreted as a Boolean, any nonempty list evaluates to true; the empty list evaluates to false. This is similar to the interpretation of integers in C as Boolean values; zero evaluates to false and any nonzero value evaluates to true. In the interest of readability, all of our example predicate functions in this chapter return #F, rather than (). The NOT function is used to invert the logic of a Boolean expression.

<i>Function</i>	<i>Meaning</i>
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
EVEN?	Is it an even number?
ODD?	Is it an odd number?
ZERO?	Is it zero?

```
(display (even? 6))  
Output: #t
```



Output Functions

- Scheme uses three different constructs for control flow:
 - One similar to the selection construct of the imperative languages
 - Two based on the evaluation control used in mathematical functions
- Scheme two-way selector function, named IF, has three parameters: a predicate expression, a then expression, and an else expression
- A call to IF has the form:

(IF predicate then_expression else_expression)

```
(DEFINE (factorial n)
  (IF (<= n 1)
      1
      (* n (factorial (- n 1))))
  ))
```

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4))))
```

- Third Scheme control mechanism is recursion, which is used, as in mathematics, to specify repetition



List Functions

- Suppose we have a function that has two parameters, an atom and a list, and the purpose of the function is to determine whether the given atom is in the given list
- Neither the atom nor the list should be evaluated; they are literal data to be examined
- To avoid evaluating a parameter, it is first given as a parameter to the primitive function QUOTE, which simply returns it without change

(QUOTE A) returns A

(QUOTE (A B C)) returns (A B C)

(CAR '(A B C)) returns A

(CAR '((A B) C D)) returns (A B)

(CAR 'A) is an error because A is not a list

(CAR '(A)) returns A

(CAR '()) is an error

(CDR '(A B C)) returns (B C)

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

(CDR 'A) is an error

(CDR '(A)) returns ()

(CDR '()) is an error

(define (**second** a_list) (**car** (**cdr** a_list)))

(display (**second** '(A B C)))

Output: B



List Functions

- Some of the most commonly used functional compositions in Scheme are built in as single functions
- For example:
- **(CAAR x)** is equivalent to **(CAR(CAR x))**
- **(CADR x)** is equivalent to **(CAR (CDR x))**
- **(CADDAR x)** is equivalent to **(CAR (CDR (CDR (CDR x))))**
- Any combination of A's and D's, up to four, are legal between the 'C' and the 'R' in the function's name

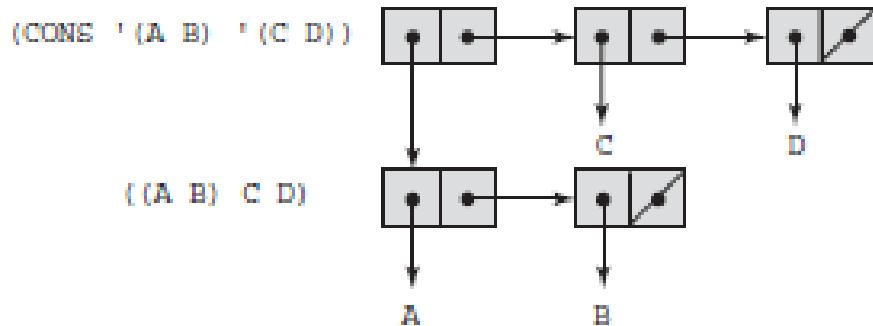
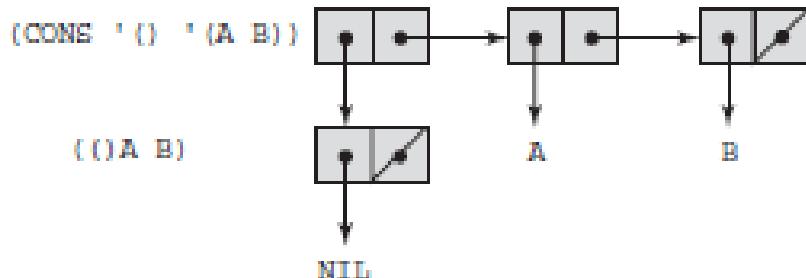
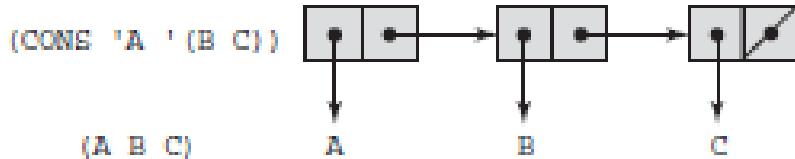
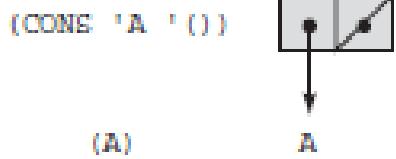
```
(CADDAR ' ((A B (C) D) E)) =  
(CAR (CDR (CDR (CAR ' ((A B (C) D) E)))) ) =  
(CAR (CDR (CDR ' (A B (C) D)))) ) =  
(CAR (CDR ' (B (C) D))) ) =  
(CAR ' ((C) D)) ) =  
(C)
```



List Functions

- Following are example calls to CONS:
(CONS 'A '()) returns (A)
(CONS 'A '(B C)) returns (A B C)
(CONS '() '(A B)) returns () A B)
(CONS '(A B) '(C D)) returns ((A B) C D)
- Note that CONS is, in a sense, the inverse of CAR and CDR
- CAR and CDR take a list apart, and CONS constructs a new list from given list parts
- The two parameters to CONS become the CAR and CDR of the new list
- Thus, if a_list is a list, then (CONS (CAR a_list) (CDR a_list)) returns a list with the same structure and same elements as a_list
- If the result of (CONS 'A 'B) is displayed, it would appear as (A . B)
 - This dotted pair indicates that instead of an atom and a pointer or a pointer and a pointer, this cell has two atoms
- LIST is a function that constructs a list from a variable number of parameters
- It is a shorthand version of nested CONS functions, as illustrated in the following: (LIST 'apple 'orange 'grape) **returns (apple orange grape)**
- Using CONS, the call to LIST above is written as follows:
(CONS 'apple (CONS 'orange (CONS 'grape '()))))

List Functions



Predicate Functions for Symbolic Atoms and Lists



- Scheme has three fundamental predicate functions, EQ?, NULL?, and LIST?, for symbolic atoms and lists
- The EQ? function takes two expressions as parameters, although it is usually used with two symbolic atom parameters
- It returns #T if both parameters have the same pointer value—that is, they point to the same atom or list; otherwise, it returns #F
- If the two parameters are symbolic atoms, EQ? returns #T if they are the same symbols; otherwise #F

```
(EQ? 'A 'A) returns #T  
(EQ? 'A 'B) returns #F  
(EQ? 'A '(A B)) returns #F  
((EQ? '(A B) '(A B))) returns #F or #T  
(EQ? 3.4 (+ 3 0.4)) returns #F or #T
```

- As the fourth example indicates, the result of comparing lists with EQ? is not consistent
- The reason for this is that two lists that are exactly the same often are not duplicated in memory
- At the time the Scheme system creates a list, it checks to see whether there is already such a list
- If there is, the new list is nothing more than a pointer to the existing list. In these cases, the two lists will be judged equal by EQ?
- However, in some cases, it may be difficult to detect the presence of an identical list, in which case a new list is created -> In this scenario, EQ? yields #F



Predicate Functions for Symbolic Atoms and Lists

- As we have seen, EQ? works for symbolic atoms but does not necessarily work for numeric atoms
- The = predicate works for numeric atoms but not symbolic atoms
- As discussed previously, EQ? also does not work reliably for list parameters
- Sometimes it is convenient to be able to test two atoms for equality when it is not known whether they are symbolic or numeric
- For this purpose, Scheme has a different predicate, EQV?, which works on both numeric and symbolic atoms

```
(EQV? 'A 'A) returns #T
(EQV? 'A 'B) returns #F
(EQV? 3 3) returns #T
(EQV? 'A 3) returns #F
(EQV? 3.4 (+ 3 0.4)) returns #T
(EQV? 3.0 3) returns #F
```

- Notice that the last example demonstrates that floating-point values are different from integer values
- EQV? is not a pointer comparison, it is a value comparison
- primary reason to use EQ? or = rather than EQV? when it is possible is that EQ? and = are faster than EQV?



Predicate Functions for Symbolic Atoms and Lists

- LIST? predicate function returns #T if its single argument is a list and #F otherwise, as in the following examples:

```
(LIST? ' (X Y) ) returns #T  
(LIST? 'X) returns #F  
(LIST? ' () ) returns #T
```

- The NULL? function tests its parameter to determine whether it is the empty list and returns #T if it is
- Consider the following examples:

```
(NULL? ' (A B) ) returns #F  
(NULL? ' () ) returns #T  
(NULL? 'A) returns #F  
(NULL? ' ( ( ) ) ) returns #F
```

- The last call yields #F because the parameter is not the empty list
- Rather, it is a list containing a single element, the empty list



Example Scheme Functions

- Consider the problem of membership of a given atom in a given list that does not include sublists -> Such a list is called a simple list
- There are three cases that must be handled in the function:
 - An empty input list
 - A match between the atom and the CAR of the list
 - A mismatch between the atom and the CAR of the list, which causes the recursive call
- These three are the three parameters to COND, with the last being the default case that is triggered by an ELSE predicate
- The complete function follows:
- If the function is named member, it could be used as follows:

(**member** 'B '(A B C)) returns #T
(**member** 'B '(A C D E)) returns #F

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list))))
  ))
```



Example Scheme Functions

- As another example, consider the problem of determining whether two given lists are equal
- If the two lists are simple, the solution is relatively easy, although some programming techniques with which the reader may not be familiar are involved
- A predicate function, `equalsimp`, for comparing simple lists is shown here:

```
(DEFINE (equalsimp list1 list2)
  (COND
    ( (NULL? list1) (NULL? list2) )
    ( (NULL? list2) #F)
    ( (EQ? (CAR list1) (CAR list2))
      (equalsimp (CDR list1) (CDR list2)) )
    (ELSE #F)
  ))
```

(display (equalsimp '(a b c) '(a b d)))

- `(append '(A B) '(C D R))` returns `(A B C D R)`
- `(append '((A B) C) '(D (E F)))` returns `((A B) C D (E F))`



Example Scheme Functions

- Another commonly needed list operation is that of constructing a new list that contains all of the elements of two given list arguments
- This is usually implemented as a Scheme function named `append`

```
(DEFINE (append list1 list2)
  (COND
    ( (NULL? list1) list2)
    (ELSE (CONS (CAR list1) (append (CDR list1) list2))))
  ))
```

(append '(A B) '(C D R)) returns (A B C D R)

(append '((A B) C) '(D (E F))) returns ((A B) C D (E F))



Example Scheme Functions

```
(DEFINE (guess list1 list2)
  (COND
    ((NULL? list1) '())
    ((member (CAR list1) list2)
     (CONS (CAR list1) (guess (CDR list1) list2)))
    (ELSE (guess (CDR list1) list2)))
  ))
```



```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list))))
  ))
```



Example Scheme Functions

```
(DEFINE (guess list1 list2)
  (COND
    ((NULL? list1) '())
    ((member (CAR list1) list2)
     (CONS (CAR list1) (guess (CDR list1) list2)))
    (ELSE (guess (CDR list1) list2)))
  ))  
  
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list))))
  ))
```

- `guess` yields a simple list that contains the common elements of its two parameter lists
- So, if the parameter lists represent sets, `guess` computes a list that represents the intersection of those two sets



LET

- LET is a function that creates a local scope in which names are temporarily bound to the values of expressions
- It is often used to factor out the common subexpressions from more complicated expressions
- These names can then be used in the evaluation of another expression, but they cannot be rebound to new values in LET
- The mathematical definitions of the real (as opposed to complex) roots of the quadratic equation $ax^2 + bx + c$ are as follows:

$$\text{root1} = \frac{(-b + \sqrt{b^2 - 4ac})}{2a} \text{ and } \text{root2} = \frac{(-b - \sqrt{b^2 - 4ac})}{2a}$$

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
    )
  (LIST (+ minus_b_over_2a root_part_over_2a)
        (- minus_b_over_2a root_part_over_2a))
```

- Because the names bound in the first part of a LET construct cannot be changed in the following expression, they are not the same as local variables in a block in an imperative language



LET

- LET is actually shorthand for a LAMBDA expression applied to a parameter
- The following two expressions are equivalent:

(**LET** ((**alpha** **7**))(* 5 **alpha**))

((**LAMBDA** (**alpha**) (* 5 **alpha**)) **7**)

- In the first expression, 7 is bound to alpha with LET
- In the second, 7 is bound to alpha through the parameter of the LAMBDA expression



Tail Recursion in Scheme

- A function is tail recursive if its recursive call is the last operation in the function
- This means that the return value of the recursive call is the return value of the nonrecursive call to the function

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list))))
```

- This function can be automatically converted by a compiler to use iteration, resulting in faster execution than in its recursive form
- However, many functions that use recursion for repetition are not tail recursive
- Programmers who were concerned with efficiency have discovered ways to rewrite some of these functions so that they are tail recursive
- One example of this uses an accumulating parameter and a helper function



Tail Recursion in Scheme

```
(DEFINE (factorial n)
  (IF (<= n 1)
      1
      (* n (factorial (- n 1)))))

})
```

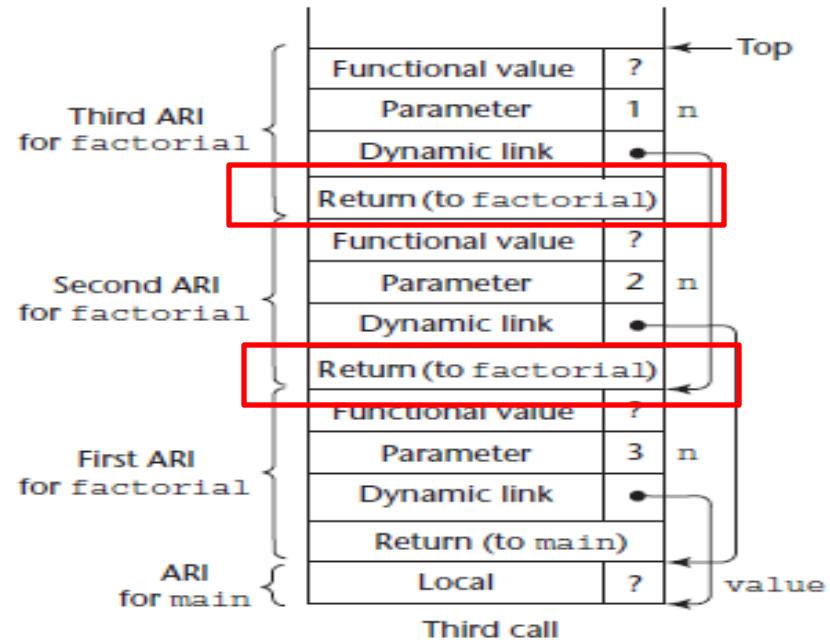
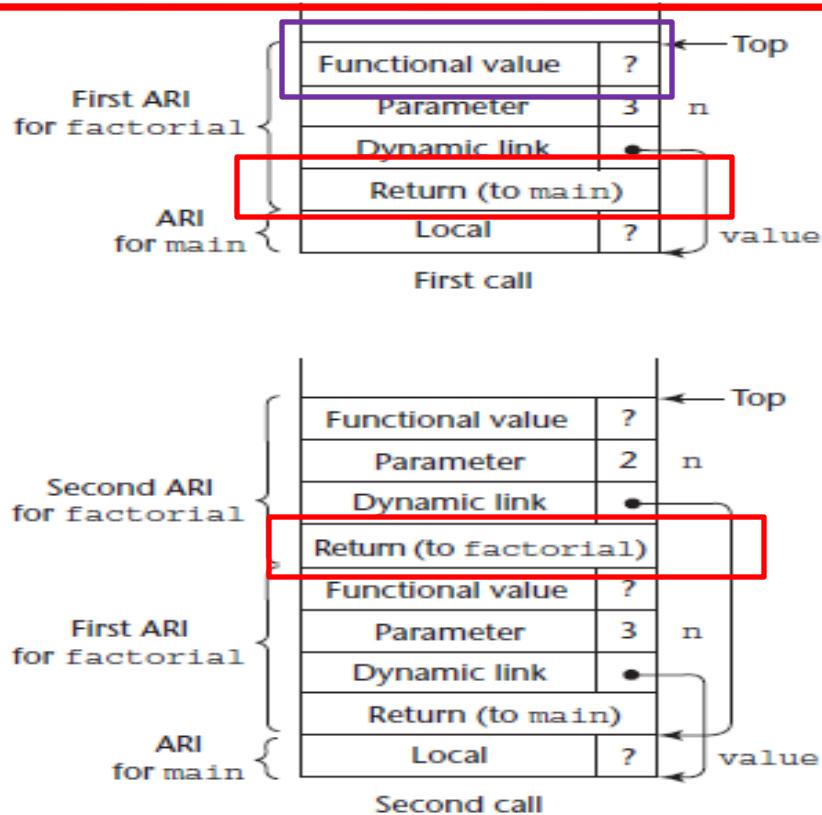
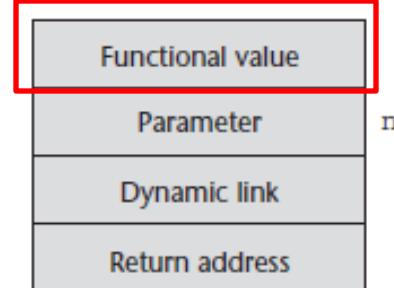
- The last operation of this function is the multiplication
- The function works by creating the list of numbers to be multiplied together and then doing the multiplications as the recursion unwinds to produce the result
- Each of these numbers is created by an activation of the function and each is stored in an activation record instance
- As the recursion unwinds the numbers are multiplied together
- This factorial function can be rewritten with an auxiliary helper function, which uses a parameter to accumulate the partial factorial
- The helper function, which is tail recursive, also takes factorial's parameter

Miscellaneous

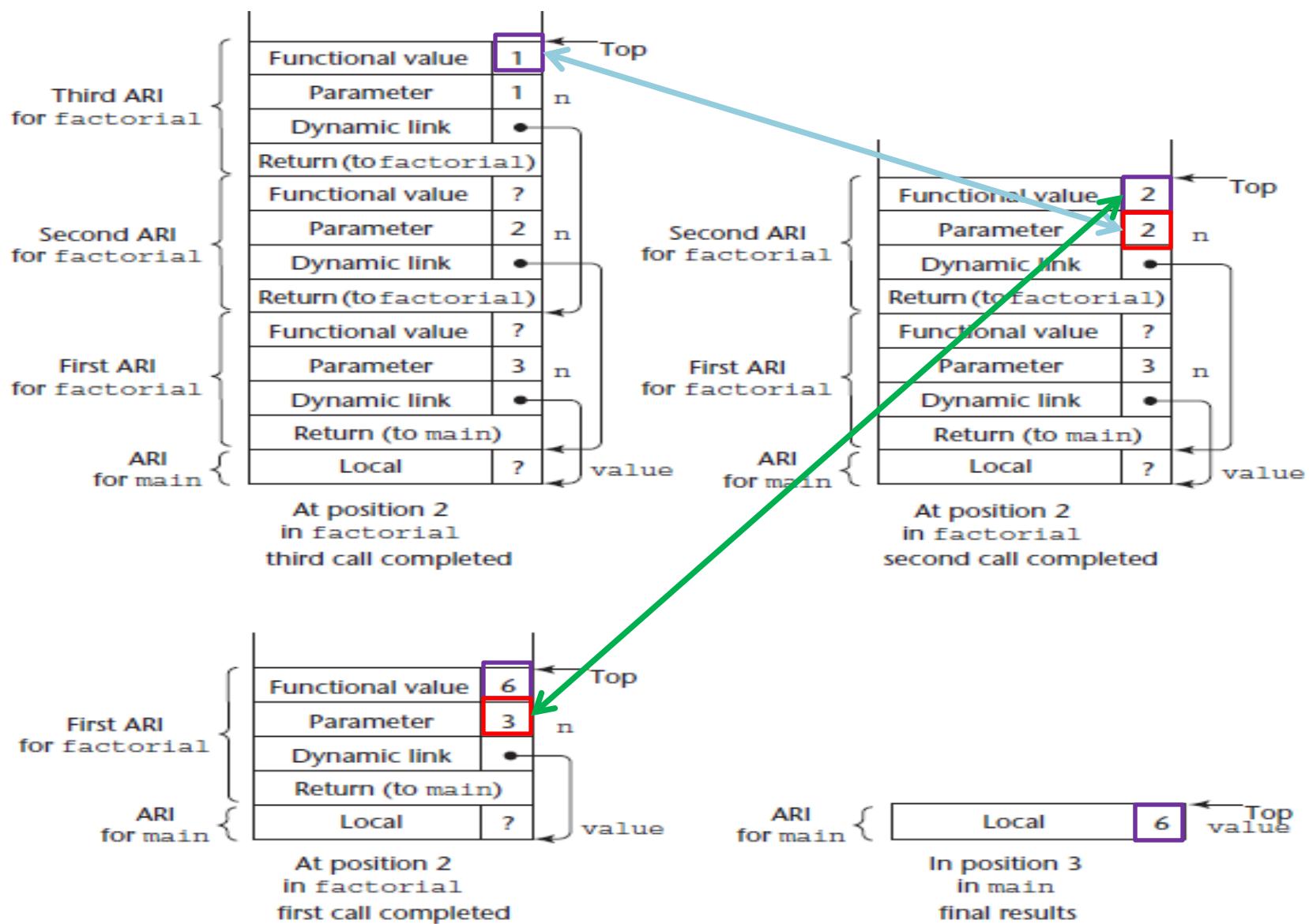
With Recursion

```

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
}
  
```



Miscellaneous





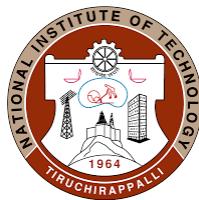
Tail Recursion in Scheme

```
(DEFINE (facthelper n factpartial)
  (IF (<= n 1)
      factpartial
      (facthelper (- n 1) (* n factpartial)))
  )
(DEFINE (factorial n)
  (facthelper n 1)
)
```

- With these functions, the result is computed during the recursive calls, rather than as the recursion unwinds
- Because there is nothing useful in the activation record instances, they are not necessary
- Regardless of how many recursive calls are requested, only one activation record instance is necessary
- This makes the tail-recursive version far more efficient than the non-tail-recursive version
- The Scheme language definition requires that Scheme language processing systems convert all tail-recursive functions to replace that recursion with iteration
- Therefore, it is important, at least for efficiency's sake, to define functions that use recursion to specify repetition to be tail recursive
- Some optimizing compilers for some functional languages can even perform conversions of some non-tail-recursive functions to equivalent tail-recursive functions and then code these functions to use iteration instead of recursion for repetition



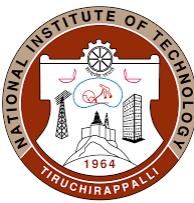
Thank You



CSPC31: Principles of Programming Languages

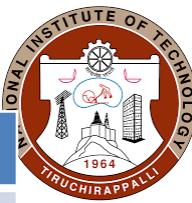
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 16 – Logic Programming Languages



Introduction

- Expresses programs in a form of symbolic logic and use a logical inferencing process to produce results
- Logic programs are declarative rather than procedural, which means that only the specifications of the desired results are stated rather than detailed procedures for producing them
- Programs in logic programming languages are collections of facts and rules
- Such a program is used by asking it questions, which it attempts to answer by consulting the facts and rules
- Programming that uses a form of symbolic logic as a programming language is often called logic programming, and languages based on symbolic logic are called logic programming languages, or declarative languages
- We have chosen to describe the logic programming language Prolog, because it is the only widely used logic language
- The syntax of logic programming languages is remarkably different from that of the imperative and functional languages
- The semantics of logic programs also bears little resemblance to that of imperative-language programs



Predicate Calculus

Universal Quantifier

$$\forall X. (\text{woman}(X) \supset \text{human}(X))$$

Existential Quantifier

$$\exists X. (\text{mother}(\text{mary}, X) \wedge \text{male}(X))$$

- The first of these propositions means that for any value of X, if X is a woman, then X is a human
- The second means that there exists a value of X such that mary is the mother of X and X is a male; in other words, mary has a son

Name	Symbol	Example	Meaning
negation	\neg	$\neg a$	not a
conjunction	\wedge	$a \wedge b$	a and b
disjunction	\vee	$a \vee b$	a or b
equivalence	$=$	$a = b$	a is equivalent to b
implication	\supset	$a \supset b$	a implies b
	\subset	$a \subset b$	b implies a

The \neg operator has the highest precedence. The operators \wedge , \vee , and $=$ all have higher precedence than \supset and \subset .



Clausal Form

- One problem with predicate calculus as we have described it thus far is that there are too many different ways of stating propositions that have the same meaning -> Great deal of redundancy
- This is not such a problem for logicians, but if predicate calculus is to be used in an automated (computerized) system, it is a serious problem
- To simplify matters, a standard form for propositions is desirable
- Clausal form, which is a relatively simple form of propositions, is one such standard form
- All propositions can be expressed in clausal form
- A proposition in clausal form has the following general syntax:

$$B_1 \cup B_2 \cup \dots \cup B_n \subset A_1 \cap A_2 \cap \dots \cap A_m$$

in which the *A's and B's are terms*

- If all of the A's are true, then at least one B is true

Primary Characteristics of Clausal Form



- Existential quantifiers are not required
- Universal quantifiers are implicit in the use of variables in the atomic propositions
- No operators other than conjunction and disjunction are required
- Also, conjunction and disjunction need appear only in the order shown in the general clausal form:
 - Disjunction on the left side and conjunction on the right side
- All predicate calculus propositions can be algorithmically converted to clausal form
- The right side of a clausal form proposition is called the antecedent
- The left side is called the consequent because it is the consequence of the truth of the antecedent



Primary Characteristics of Clausal Form

$\text{likes(bob, trout)} \subset \text{likes(bob, fish)} \cap \text{fish(trout)}$

- States that “if bob likes fish and a trout is a fish, then bob likes trout”

$\text{father(louis, al)} \cup \text{father(louis, violet)} \subset$
 $\text{father(al, bob)} \cap \text{mother(violet, bob)} \cap \text{grandfather(louis, bob)}$

- States that “if al is bob’s father and violet is bob’s mother and louis is bob’s grandfather, then louis is either al’s father or violet’s father



Overview of Logic Programming

- Languages used for logic programming are called declarative languages, because programs written in them consist of declarations rather than assignments and control flow statements
- These declarations are actually statements, or propositions, in symbolic logic
- One of the essential characteristics of logic programming languages is their semantics, which is called declarative semantics
- Prolog has two basic statement forms; these correspond to the **headless** and **headed Horn** clauses of predicate calculus



Basic Elements of Prolog

- **Fact Statements**

- Our discussion of Prolog statements begins with those statements used to construct the hypotheses, or database of assumed information—the statements from which new information can be inferred
- Simplest form of **headless Horn clause** in Prolog is a single structure, which is interpreted as an unconditional assertion, or fact
- Logically, facts are simply propositions that are assumed to be true

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



Basic Elements of Prolog

- **Fact Statements**

- Our discussion of Prolog statements begins with those statements used to construct the hypotheses, or database of assumed information—the statements from which new information can be inferred
- These simple structures state certain facts about jake, shelley, bill, and mary
- For example, the first states that shelley is a female
- The last four connect their two parameters with a relationship that is named in the functor atom; for example, the fifth proposition might be interpreted to mean that bill is the father of jake
- Note that these Prolog propositions, like those of predicate calculus, have no intrinsic semantics
- They mean whatever the programmer wants them to mean
- For example, the proposition father(bill, jake). could mean bill and jake have the same father or that jake is the father of bill
- The most common and straightforward meaning, however, might be that bill is the father of jake

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



Basic Elements of Prolog

- **Rule Statements**
- **Headed Horn clause**

consequence :- antecedent_expression.

- It is read as follows: “consequence can be concluded if the antecedent expression is true or can be made to be true by some instantiation of its variables.”

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

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

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

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

Goal Statements

- So far, we have described the Prolog statements for logical propositions, which are used to describe both known facts and rules that describe logical relationships among facts
- These statements are the basis for the theorem-proving model
- The theorem is in the form of a proposition that we want the system to either prove or disprove
- In Prolog, these propositions are called goals, or queries
- The syntactic form of Prolog goal statements is identical to that of headless Horn clauses
- For example, we could have **man(fred).**, to which the system will respond either yes or no.
 - The answer yes means that the system has proved the goal was true under the given database of facts and relationships
 - The answer no means that either the goal was determined to be false or the system was simply unable to prove it



Sample Program

- Prolog always performs depth-first-search, Matches facts & rules (i.e. knowledge base) in top-down manner and resolves the goals or subgoals in left-to-right manner
- Most important thing to keep in mind while writing prolog program - "order of writing facts & rules always matters"

Facts English meanings

```
food(burger).        // burger is a food  
food(sandwich).     // sandwich is a food  
food(pizza).        // pizza is a food  
lunch(sandwich).    // sandwich is a lunch  
dinner(pizza).      // pizza is a dinner
```

Rules

```
meal(X) :- food(X).    // Every food is a meal OR Anything is a meal if it is a food
```

Queries / Goals

```
?- food(pizza).      // Is pizza a food?  
?- meal(X), lunch(X). // Which food is meal and lunch?  
?- dinner(sandwich). // Is sandwich a dinner?
```



Sample Program

- Prolog always performs depth-first-search, Matches facts & rules (i.e. knowledge base) in top-down manner and resolves the goals or subgoals in left-to-right manner
- Most important thing to keep in mind while writing prolog program - "order of writing facts & rules always matters"

Facts

	English meanings
studies(charlie, csc135).	// charlie studies csc135
studies(olivia, csc135).	// olivia studies csc135
studies(jack, csc131).	// jack studies csc131
studies(arthur, csc134).	// arthur studies csc134
teaches(kirke, csc135).	// kirke teaches csc135
teaches(collins, csc131).	// collins teaches csc131
teaches(collins, csc171).	// collins teaches csc171
teaches(juniper, csc134).	// juniper teaches csc134

Rules

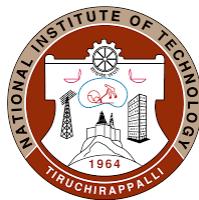
professor(X, Y) :- teaches(X, C), studies(Y, C). // X is a professor of Y if X teaches C and Y studies C.

Queries / Goals

?- food(pizza). // Is pizza a food?
?- meal(X), lunch(X). // Which food is meal and lunch?
?- dinner(sandwich). // Is sandwich a dinner?



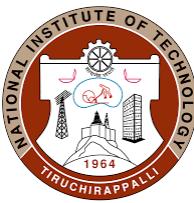
Thank You



CSPC31: Principles of Programming Languages

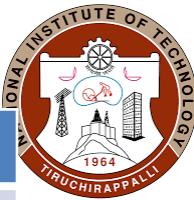
Ph: 999 470 4853

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
E-Mail: balakrishnan@nitt.edu



Books

- **Text Books**
 - ✓ Robert W. Sebesta, "*Concepts of Programming Languages*", Tenth Edition, Addison Wesley, 2012.
 - ✓ Michael L. Scott, "*Programming Language Pragmatics*", Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
 - ✓ Allen B Tucker, and Robert E Noonan, "*Programming Languages – Principles and Paradigms*", Second Edition, Tata McGraw Hill, 2007.
 - ✓ R. Kent Dybvig, "*The Scheme Programming Language*", Fourth Edition, MIT Press, 2009.
 - ✓ Jeffrey D. Ullman, "*Elements of ML Programming*", Second Edition, Prentice Hall, 1998.
 - ✓ Richard A. O'Keefe, "*The Craft of Prolog*", MIT Press, 2009.
 - ✓ W. F. Clocksin, C. S. Mellish, "*Programming in Prolog: Using the ISO Standard*", Fifth Edition, Springer, 2003.



Chapters

Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 14 – Exception Handling and Event Handling



Objectives

- Describes the fundamental concepts of exception handling
- Introduction to the design issues for exception handling
- Description and evaluation of exception-handling facilities of C++
- Event handling



Miscellaneous

```
int main()
{
    int x = -1;
    cout << "Before try \n";
    try
    {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
        }
        cout << "After throw (Never executed) \n";
    }
    catch (int x)
    {
        cout << "Exception Caught \n";
    }
    cout << "After catch (Will be executed) \n";
    return 0;
}
```

O/P: Before try
Inside try
Exception Caught
After catch (Will be executed)

```
int main()
{
    try
    {
        throw 10;
    }
    catch (char *excp)
    {
        cout << "Caught " << excp;
    }
    catch (...)
    {
        cout << "Default Exception\n";
    }
    return 0;
}
```

O/P: Default Exception



Miscellaneous

```
int main()
{
    try
    {
        throw 'a';
    }
    catch (int x)
    {
        cout << "Caught " << x;
    }
    →catch (...)
    {
        cout << "Default Exception\n";
    }
    return 0;
}
```

O/P: Default Exception

```
int main()
{
    try
    {
        throw 'a';
    }
    catch (int x)
    {
        cout << "Caught ";
    }
    return 0;
}
```

If an exception is thrown and not caught anywhere, the program terminates abnormally

O/P: terminate called after throwing an instance of 'char'
This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information. 7



Miscellaneous

// This function signature is fine by the compiler, but not recommended. Ideally, **the function should specify all uncaught exceptions and function signature should be "void fun(int *ptr, int x) throw (int *, int)"**

```
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
}
int main()
{
    try
    {
        fun(NULL, 0);
    }
    catch(...)
    {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not. **Eg:** In C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. **Eg:** The following program compiles fine, but ideally signature of fun() should list unchecked exceptions

O/P: Caught exception from fun()

// Here we specify the exceptions that this function throws.

```
void fun(int *ptr, int x) throw (int *, int)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
}
int main()
{
    try
    {
        fun(NULL, 0);
    }
    catch(...)
    {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

O/P: Caught exception from fun()



Miscellaneous

```
import java.io.*;  
class Main  
{  
    public static void main(String[] args)  
    {  
        FileReader file = new  
        FileReader("C:\\test\\a.txt");  
→ BufferedReader fileInput = new  
        BufferedReader(file);  
        for (int counter = 0; counter < 3;  
        counter++)  
            System.out.println(fileInput.readLine());  
        fileInput.close();  
    }  
}  
O/P: Exception in thread "main"  
java.lang.RuntimeException: Uncompilable source  
code - unreported exception  
java.io.FileNotFoundException; must be caught or  
declared to be thrown at Main.main(Main.java:5)
```

```
import java.io.*;  
class Main  
{  
    public static void main(String[] args)  
throws IOException  
    {  
        FileReader file = new  
        FileReader("C:\\test\\a.txt");  
→ BufferedReader fileInput = new  
        BufferedReader(file);  
        // Print first 3 lines of file "C:\\test\\a.txt"  
        for (int counter = 0; counter < 3;  
        counter++)  
            System.out.println(fileInput.readLine());  
        fileInput.close();  
    }  
}
```

Note: To fix the problem, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the above program. Since **FileNotFoundException** is a subclass of **IOException**, we can just specify **IOException** in the throws list and make the above program compiler-error-free



Miscellaneous

```
class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    try
    {
        // Some monitored code
        throw d;
    }
    →catch(Base b)
    {
        cout<<"Caught Base Exception";
    }
    catch(Derived d)
    { //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
    }
    getchar();
    return 0;
}
```

A derived class exception should be caught before a base class exception

O/P: Caught Base Exception

```
class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    try
    {
        // Some monitored code
        throw d;
    }
    →catch(Derived d)
    {
        cout<<"Caught Derived Exception";
    }
    catch(Base b)
    {
        cout<<"Caught Base Exception";
    }
    getchar();
    return 0;
}
```

O/P: Caught Derived Exception



Miscellaneous

```
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        →catch (int n)
        {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    →catch (int n)
    {
        cout << "Handle remaining ";
    }
    return 0;
}
O/P: Handle Partially Handle remaining
```

In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw;"

```
class Test {
public:
    Test()
    {
        cout << "Constructor of Test " << endl;
    }
    ~Test()
    {
        cout << "Destructor of Test " << endl;
    }
};
```

When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block

```
int main()
{
    try
    {
        Test t1;
        throw 10;
    }
    →catch (int i)
    {
        cout << "Caught " << i << endl;
    }
}
```

O/P: Constructor of Test
Destructor of Test
Caught 10

Miscellaneous

Event Handling

```

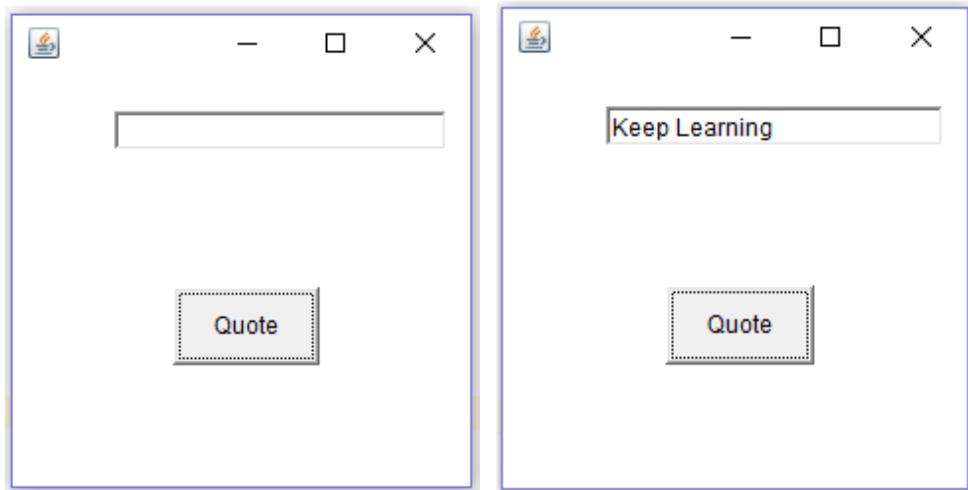
import java.awt.*;
import java.awt.event.*;
class EventHandle extends Frame
    implements ActionListener
{
    TextField textField;
    EventHandle()
    {
        textField = new TextField();
        textField.setBounds(60,50,170,20);
        Button button = new Button("Quote");
        button.setBounds(90,140,75,40);
        //1
        button.addActionListener(this);
        add(button);
        add(textField);
        setSize(250,250);
        setLayout(null);
        setVisible(true);
    }
}

```

```

//2
public void actionPerformed(ActionEvent e)
{
    textField.setText("Keep Learning");
}
public static void main(String args[])
{
    new EventHandle();
}

```





Introduction

- Most computer hardware systems are capable of detecting certain run-time error conditions, such as floating-point overflow
- Early programming languages were designed and implemented in such a way that the user program could neither detect nor attempt to deal with such errors
- In these languages, the occurrence of such an error simply causes the program to be terminated and control to be transferred to the operating system
- The typical operating system reaction to a run-time error is to display a diagnostic message, which may be meaningful and therefore useful, or highly cryptic
- After displaying the message, the program is terminated
- In the case of input and output operations, however, the situation is somewhat different
- Eg: A Fortran Read statement can intercept input errors and end-of-file conditions, both of which are detected by the input device hardware
- In both cases, the Read statement can specify the label of some statement in the user program that deals with the condition
- In the case of the end-of-file, it is clear that the condition is not always considered an error
- In most cases, it is nothing more than a signal that one kind of processing is completed and another kind must begin
- In spite of the obvious difference between end-of-file and events that are always errors, such as a failed input process, Fortran handles both situations with the same mechanism

Read (Unit=5, Fmt=1000, Err=100, End=999) Weight



Introduction

Read(Unit=5, Fmt=1000, Err=100, End=999) Weight

- Fortran uses simple branches for both input errors and end-of-file
- There is a category of serious errors that are not detectable by hardware but can be detected by code generated by the compiler
- Eg: Array subscript range errors are almost never detected by hardware, but they lead to serious errors that often are not noticed until later in the program execution
- Detection of subscript range errors is sometimes required by the language design
- Eg: Java compilers usually generate code to check the correctness of every subscript expression (they do not generate such code when it can be determined at compile time that a subscript expression cannot have an out-of-range value, for example, if the subscript is a literal)
- In C, subscript ranges are not checked because the cost of such checking was (and still is) not believed to be worth the benefit of detecting such errors
- In some compilers for some languages, subscript range checking can be selected (if not turned on by default) or turned off (if it is on by default) as desired in the program or in the command that executes the compiler



Introduction

- Designers of most contemporary languages have included mechanisms that allow programs to react in a standard way to certain run-time errors, as well as other program-detected unusual events
- Programs may also be notified when certain events are detected by hardware or system software, so that they also can react to these events
- These mechanisms are collectively called exception handling
- Perhaps the most plausible reason some languages do not include exception handling is the complexity it adds to the language

Basic Concepts

- We consider both the errors detected by hardware, such as disk read errors, and unusual conditions, such as end-of-file (which is also detected by hardware), to be exceptions
- We further extend the concept of an exception to include errors or unusual conditions that are software-detectable (by either a software interpreter or the user code itself)
- Accordingly, we define exception to be any unusual event, erroneous or not, that is detectable by either hardware or software and that may require special processing



Introduction

- The special processing that may be required when an exception is detected -> Exception Handling
- This processing is done by a code unit or segment -> Exception Handler
- An exception is raised when its associated event occurs
- In some C-based languages, exceptions are said to be thrown, rather than raised
- Different kinds of exceptions require different exception handlers
- Detection of end-of-file nearly always requires some specific program action
- But, clearly, that action would not also be appropriate for an array index range error exception
- In some cases, the only action is the generation of an error message and an orderly termination of the program
- In some situations, it may be desirable to ignore certain hardware-detectable exceptions—for example, division by zero—for a time
- This action would be done by disabling the exception
- A disabled exception could be enabled again at a later time



Introduction

- The absence of separate or specific exception-handling facilities in a language does not preclude the handling of user-defined, software-detected exceptions
- Such an exception detected within a program unit is often handled by the unit's caller, or invoker
- **One possible design** is to send an auxiliary parameter, which is used as a status variable
- The status variable is assigned a value in the called subprogram according to the correctness and/or normalness of its computation
- Immediately upon return from the called unit, the caller tests the status variable
- If the value indicates that an exception has occurred, the handler, which may reside in the calling unit, can be enacted
- Many of the **C standard library functions** use a variant of this approach: The return values are used as error indicators. **Eg:** int main()



Introduction

- **Another possibility** is to pass a label parameter to the subprogram
- Of course, this approach is possible only in languages that allow labels to be used as parameters
- Passing a label allows the called unit to return to a different point in the caller if an exception has occurred
- As in the first alternative, the handler is often a segment of the calling unit's code
- This is a common use of label parameters in **Fortran**
- **A third possibility** is to have the handler defined as a separate subprogram whose name is passed as a parameter to the called unit
- In this case, the handler subprogram is provided by the caller, but the called unit calls the handler when an exception is raised
- **One problem** with this approach is that one is required to send a handler subprogram with every call to every subprogram that takes a handler subprogram as a parameter, whether it is needed or not
- Furthermore, to deal with several different kinds of exceptions, several different handler routines would need to be passed, complicating the code



Introduction

- If it is desirable to handle an exception in the unit in which it is detected, the handler is included as a segment of code in that unit
- There are some definite advantages to having exception handling built into a language
- First, without exception handling, the code required to detect error conditions can considerably clutter a program
- **Eg:** Suppose a subprogram includes expressions that contain 10 references to elements of a matrix named mat, and any one of them could have an index out-of-range error. Further suppose that the language does not require index range checking
- Without built-in index range checking, every one of these operations may need to be preceded by code to detect a possible index range error
- **Eg:** Consider the following reference to an element of mat, which has 10 rows and 20 columns

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
    sum += mat [row] [col];
else
    System.out.println("Index range error on mat, row = " +
                       row + " col = " + col);
```



Introduction

- The presence of exception handling in the language would permit the compiler to insert machine code for such checks before every array element access, greatly shortening and simplifying the source program
- Another advantage of language support for exception handling results from exception propagation
- Exception propagation allows an exception raised in one program unit to be handled in some other unit in its dynamic or static ancestry
- This allows a single exception handler to be used for any number of different program units
- This reuse can result in significant savings in development cost, program size, and program complexity
- A language that supports exception handling encourages its users to consider all of the events that could occur during program execution and how they can be handled
- This approach is far better than not considering such possibilities and simply hoping nothing will go wrong



Introduction

Design Issues

- System might allow both predefined and user-defined exceptions and exception handlers
- Note that predefined exceptions are implicitly raised, whereas user-defined exceptions must be explicitly raised by user code
- Consider the following skeletal subprogram that includes an exception-handling mechanism for an implicitly raised exception:

```
void example() {  
    ...  
    average = sum / total;  
    ...  
    return;  
    /* Exception handlers */  
    when zero_divide {  
        average = 0;  
        printf("Error-divisor (total) is zero\n");  
    }  
    ...  
}
```

- The exception of division by zero, which is implicitly raised, causes control to transfer to the appropriate handler, which is then executed



Introduction

- First design issue for exception handling is how an exception occurrence is bound to an exception handler
- This issue occurs on two different levels
- On the unit level, there is the question of how the same exception being raised at different points in a unit can be bound to different handlers within the unit
- **Eg:** In the example subprogram, there is a handler for a division-by-zero exception that appears to be written to deal with an occurrence of division by zero in a particular statement (the one shown)
- But suppose the function includes several other expressions with division operators
- For those operators, this handler would probably not be appropriate
- So, it should be possible to bind the exceptions that can be raised by particular statements to particular handlers, even though the same exception can be raised by many different statements



Introduction

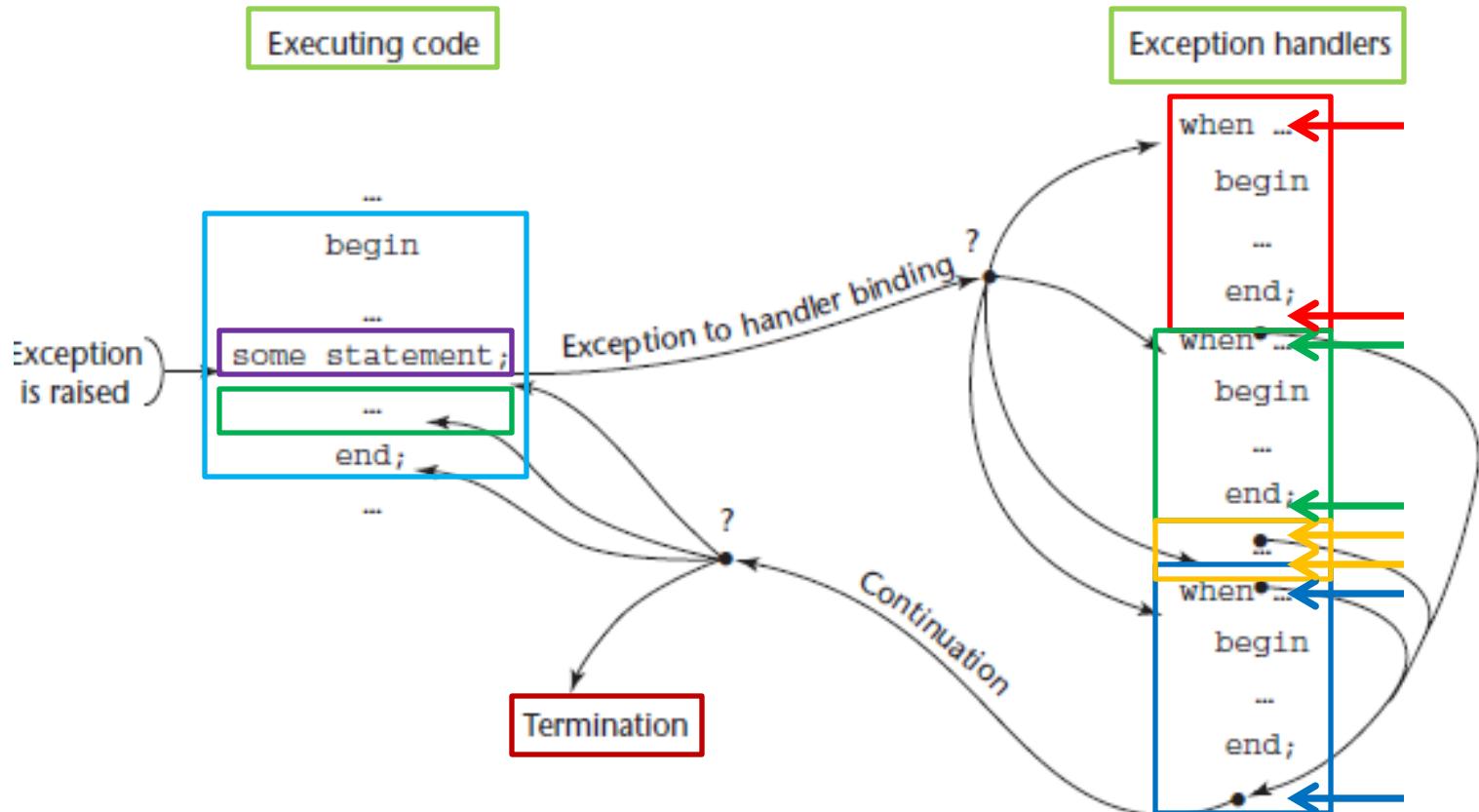
- At a higher level, the binding question arises when there is no exception handler local to the unit in which the exception is raised
- In this case, the language designer must decide whether to propagate the exception to some other unit and, if so, where
- How this propagation takes place and how far it goes have an important impact on the writability of exception handlers
- Eg: If handlers must be local, then many handlers must be written, which complicates both the writing and reading of the program
- On the other hand, if exceptions are propagated, a single handler might handle the same exception raised in several program units, which may require the handler to be more general than one would prefer
- An issue that is related to the binding of an exception to an exception handler is whether information about the exception is made available to the handler



Introduction

- After an exception handler executes, either control can transfer to somewhere in the program outside of the handler code or program execution can simply terminate
- We term this the question of control continuation after handler execution, or simply continuation
- Termination is obviously the simplest choice, and in many error exception conditions, the best
- However, in other situations, particularly those associated with unusual but not erroneous events, the choice of continuing execution is best
- This design is called resumption
- In these cases, some conventions must be chosen to determine where execution should continue
- It might be the statement that raised the exception, the statement after the statement that raised the exception, or possibly some other unit
- The choice to return to the statement that raised the exception may seem like a good one, but in the case of an error exception, it is useful only if the handler somehow is able to modify the values or operations that caused the exception to be raised
- Otherwise, the exception will simply be reraised
- The required modification for an error exception is often very difficult to predict
- Even when possible, however, it may not be a sound practice
- It allows the program to remove the symptom of a problem without removing the cause

Introduction





Introduction

- When exception handling is included, a subprogram's execution can terminate in two ways
 - When its execution is complete or when it encounters an exception
 - In some situations, it is necessary to complete some computation regardless of how subprogram execution terminates
 - The ability to specify such a computation is called *finalization*
 - The choice of whether to support finalization is obviously a design issue for exception handling
- Another design issue is the following
 - If users are allowed to define exceptions, how are these exceptions specified? The usual answer is to require that they be declared in the specification parts of the program units in which they can be raised. The scope of a declared exception is usually the scope of the program unit that contains the declaration



Introduction

- In the case where a language provides predefined exceptions, several other design issues follow. For example, should the language run-time system provide default handlers for the built-in exceptions, or should the user be required to write handlers for all exceptions? Another question is whether predefined exceptions can be raised explicitly by the user program. This usage can be convenient if there are software-detectable situations in which the user would like to use a predefined handler
- Another issue is whether hardware-detectable errors can be handled by user programs. If not, all exceptions obviously are software detectable. A related question is whether there should be any predefined exceptions. Predefined exceptions are implicitly raised by either hardware or system software
- Finally, there is the question of whether exceptions, either predefined or user defined, can be temporarily or permanently disabled. This question is somewhat philosophical, particularly in the case of predefined error conditions
- For example, suppose a language has a predefined exception that is raised when a subscript range error occurs. Many believe that subscript range errors should always be detected, and therefore it should not be possible for the program to disable detection of these errors. Others argue that subscript range checking is too costly for production software, where, presumably, the code is sufficiently error free that range errors should not occur



Introduction

The exception-handling design issues can be summarized as follows:

- How and where are exception handlers specified, and what is their scope?
- How is an exception occurrence bound to an exception handler?
- Can information about an exception be passed to the handler?
- Where does execution continue, if at all, after an exception handler completes its execution? (This is the question of continuation or resumption)
- Is some form of finalization provided?
- How are user-defined exceptions specified?
- If there are predefined exceptions, should there be default exception handlers for programs that do not provide their own?
- Can predefined exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that may be handled?
- Are there any predefined exceptions?
- Should it be possible to disable predefined exceptions?



Exception Handling in C++

- One major difference between the exception handling of C++ and that of Ada is the absence of predefined exceptions in C++ (other than in its standard libraries)
- Thus, in C++, exceptions are user or library defined and explicitly raised

Exception Handlers

- C++ uses a special construct that is introduced with the reserved word `try` for this purpose
- A `try` construct includes a compound statement called the `try` clause and a list of exception handlers
- The compound statement defines the scope of the following handlers

```
try {
    /** Code that might raise an exception
}
catch(formal parameter) {
    /** A handler body
}
...
catch(formal parameter) {
    /** A handler body
}
```



Exception Handling in C++

- Each catch function is an exception handler
- A catch function can have only a single formal parameter, which is similar to a formal parameter in a function definition in C++, including the possibility of it being an ellipsis (...)
- A handler with an ellipsis formal parameter is the catch-all handler; it is enacted for any raised exception if no appropriate handler was found
- The formal parameter also can be a naked type specifier, such as float, as in a function prototype
 - In such a case, the only purpose of the formal parameter is to make the handler uniquely identifiable
- When information about the exception is to be passed to the handler, the formal parameter includes a variable name that is used for that purpose
- Because the class of the parameter can be any user-defined class, the parameter can include as many data members as are necessary
- In C++, exception handlers can include any C++ code

Exception Handling in C++



Binding Exceptions to Handlers

- C++ exceptions are raised only by the explicit statement `throw`, whose general form in EBNF is

`throw [expression];`

- The brackets here are metasymbols used to specify that the expression is optional
- A `throw` without an operand can appear only in a handler
- When it appears there, it reraises the exception, which is then handled elsewhere
- The type of the `throw` expression selects the particular handler, which of course must have a “matching” type formal parameter
- An exception raised in a `try` clause causes an immediate end to the execution of the code in that `try` clause
- The search for a matching handler begins with the handlers that immediately follow the `try` clause
- The matching process is done sequentially on the handlers until a match is found
- This means that if any other match precedes an exactly matching handler, the exactly matching handler will not be used
- Therefore, handlers for specific exceptions are placed at the top of the list, followed by more generic handlers
- The last handler is often one with an ellipsis (`...`) formal parameter, which matches any exception
- This would guarantee that all exceptions were caught



Exception Handling in C++

- If an exception is raised in a try clause and there is no matching handler associated with that try clause, the exception is propagated
- If the try clause is nested inside another try clause, the exception is propagated to the handlers associated with the outer try clause
- If none of the enclosing try clauses yields a matching handler, the exception is propagated to the caller of the function in which it was raised
- If the call to the function was not in a try clause, the exception is propagated to that function's caller
- If no matching handler is found in the program through this propagation process, the default handler is called

Continuation

- After a handler has completed its execution, control flows to the first statement following the try construct (the statement immediately after the last handler in the sequence of handlers of which it is an element)
- A handler may reraise an exception, using a throw without an expression, in which case that exception is propagated



Miscellaneous

```
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        →catch (int n)
        {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    →catch (int n)
    {
        cout << "Handle remaining ";
    }
    return 0;
}
O/P: Handle Partially Handle remaining
```

In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw;"

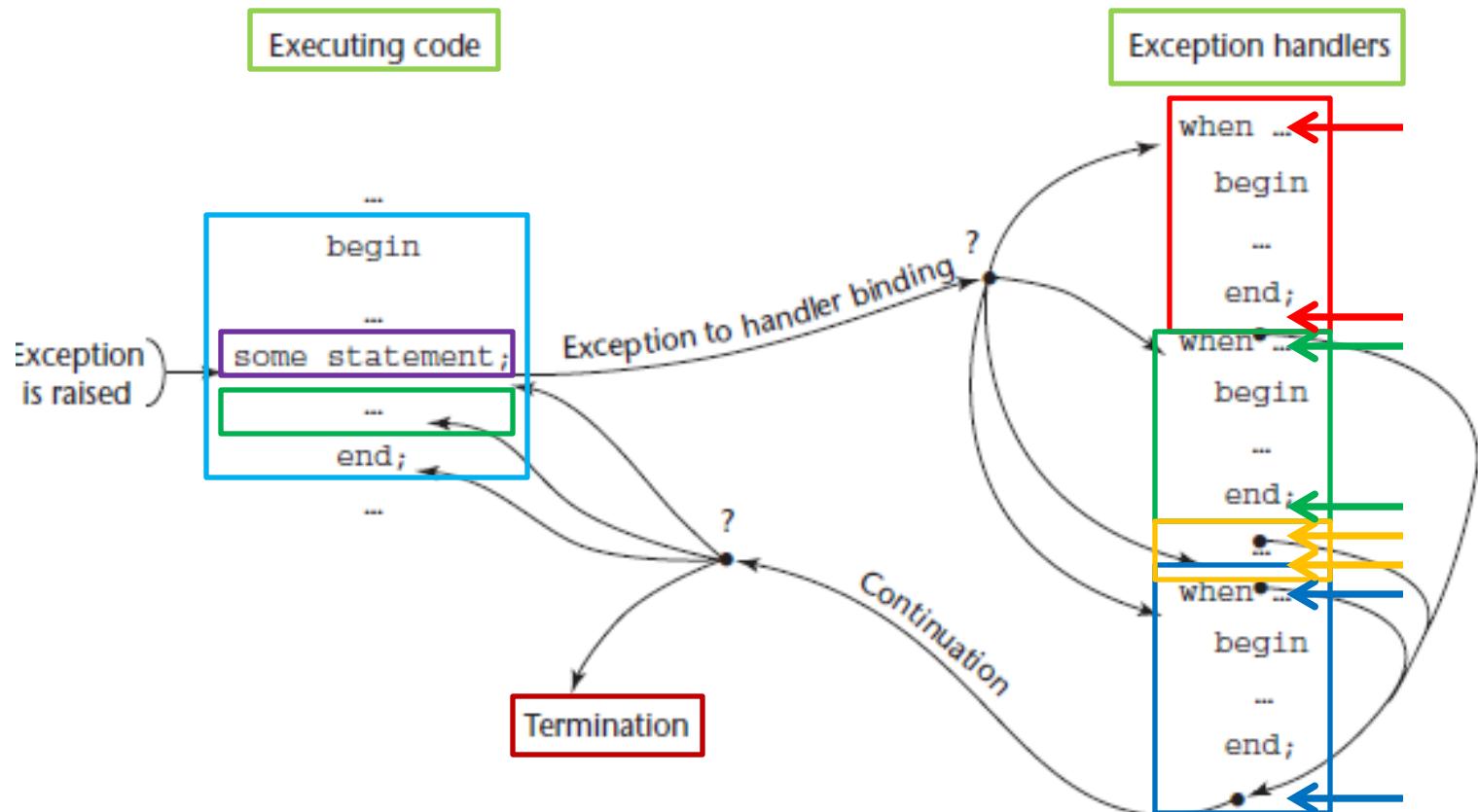
```
class Test {
public:
    Test()
    {
        cout << "Constructor of Test " << endl;
    }
    ~Test()
    {
        cout << "Destructor of Test " << endl;
    }
};
```

When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block

```
int main()
{
    try
    {
        Test t1;
        throw 10;
    }
    →catch (int i)
    {
        cout << "Caught " << i << endl;
    }
}
```

O/P: Constructor of Test
Destructor of Test
Caught 10

Introduction



Exception Handling in C++



Other Design Choices

- Exception handling of C++ is simple
- There are only user-defined exceptions, and they are not specified (though they might be declared as new classes)
- There is a default exception handler, unexpected, whose only action is to terminate the program
- This handler catches all exceptions not caught by the program
- It can be replaced by a user-defined handler
 - The replacement handler must be a function that returns void and takes no parameters
 - The replacement function is set by assigning its name to `set_terminate`
- Exceptions cannot be disabled
- A C++ function can list the types of the exceptions (the types of the `throw` expressions) that it could raise
- This is done by attaching the reserved word `throw`, followed by a parenthesized list of these types, to the function header

`int fun() throw (int, char *) { ... }`

- If the function does throw some unlisted exception, the program will be terminated



Miscellaneous

// Here we specify the exceptions that this function throws.

```
void fun(int *ptr, int x) throw (int *, char)
{
    if (ptr == NULL)
        throw 5;
    if (x == 0)
        throw 6;
}
int main()
{
    try
    {
        fun(NULL, 0);
    }
    catch(...)
    {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Program will be terminated



Exception Handling in C++

- If the types in the throw clause are classes, then the function can raise any exception that is derived from the listed classes
- If a function header has a throw clause and raises an exception that is not listed in the throw clause and is not derived from a class listed there, the default handler is called
- Note that this error cannot be detected at compile time
- The list of types in the list may be empty, meaning that the function will not raise any exceptions
- If there is no throw specification on the header, the function can raise any exception
- The list is not part of the function's type
- If a function overrides a function that has a throw clause, the overriding function cannot have a throw clause with more exceptions than the overridden function
- Although C++ has no predefined exceptions, the standard libraries define and throw exceptions, such as `out_of_range`, which can be thrown by library container classes, and `overflow_error`, which can be thrown by math library functions



Miscellaneous

```
class FileException
{
    Exception 1: FileNotFoundException
    Exception 2: End-of-File
};

void fun(int *ptr, int x) throw (int, FileException)
{
    if (ptr == NULL)
        throw Read-Only;
    if (x == 0)
        throw Write-Only;
}
```

```
int main()
{
    Derived d;
    try
    {
        fun(NULL, 0);
    }
    catch(FileException b) ←
    {
        cout<<"Caught Base Exception";
    }
    getchar();
    return 0;
}
```

If a function header has a throw clause and raises an exception that is not listed in the throw clause and is not derived from a class listed there, then the program won't be terminated. Instead, the default handler is called



Miscellaneous

// Here we specify the exceptions that this function throws.

```
void fun(int *ptr, int x) throw (int *, int)
```

```
{  
    if (ptr == NULL)  
        throw ptr;  
    if (x == 0)  
        throw x;  
}
```

```
int main()  
{
```

```
    try  
    {  
        fun(NULL, 0);  
    }
```

→ **catch(...)**

```
    {  
        cout << "Caught exception from fun()";  
    }  
    return 0;  
}
```

O/P: Caught exception from fun()

// Here we specify the exceptions that this function throws.

```
void fun(int *ptr, int x) throw ()
```

```
{  
    if (ptr == NULL)  
        throw ptr;  
    if (x == 0)  
        throw x;  
}
```

```
int main()  
{
```

```
    try  
    {  
        fun(NULL, 0);  
    }
```

→ **catch(...)**

```
    {  
        cout << "Caught exception from fun()";  
    }  
    return 0;  
}
```

Compilation Error



Miscellaneous

```
class demo {  
};  
int main()  
{  
    try  
    {  
        throw demo();  
    }  
    catch (demo d)  
    {  
        cout << "Caught exception of demo  
class \n";  
    }  
}
```

O/P: Caught exception of demo class

```
class demo1 {  
};  
  
class demo2 {  
};  
  
int main()  
{  
    for (int i = 1; i <= 2; i++) {  
        try  
        {  
            if (i == 1)  
                throw demo1();  
  
            else if (i == 2)  
                throw demo2();  
        }  
        catch (demo1 d1)  
        {  
            cout << "Caught exception of demo1 class  
\n";  
        }  
  
        catch (demo2 d2)  
        {  
            cout << "Caught exception of demo2 class  
\n";  
        }  
    }  
}
```

O/P: Caught exception of demo1 class
Caught exception of demo2 class



Miscellaneous

```
class demo1 {  
};  
  
class demo2 : public demo1 {  
};  
  
int main()  
{  
    for (int i = 1; i <= 2; i++) {  
        try  
        {  
            if (i == 1)  
                throw demo1();  
  
            else if (i == 2)  
                throw demo2();  
        }  
        catch (demo1 d1)  
        {  
            cout << "Caught exception of demo1 class \n";  
        }  
        catch (demo2 d2)  
        {  
            cout << "Caught exception of demo2 class \n";  
        }  
    }  
}
```

O/P: Caught exception of demo1 class
Caught exception of demo1 class

```
class demo {  
    int num;  
  
public:  
    demo(int x)  
    {  
        try {  
            if (x == 0)  
                // catch block would be called  
                throw "Zero not allowed ";  
            num = x;  
            show();  
        }  
        catch (const char* exp) {  
            cout << "Exception caught \n ";  
            cout << exp << endl;  
        }  
        void show()  
        {  
            cout << "Num = " << num << endl;  
        }  
    }  
  
    int main()  
    {  
        // constructor will be called  
        demo(0);  
        cout << "Again creating object \n";  
        demo(1);  
    }  
}
```

O/P: Exception caught
Zero not allowed
Again creating object
Num = 1



Exception Handling in C++

Evaluation

- Unhandled exceptions in functions are propagated to the function's caller
- There are no predefined hardware-detectable exceptions that can be handled by the user, and exceptions are not named
- Exceptions are connected to handlers through a parameter type in which the formal parameter may be omitted
- The type of the formal parameter of a handler determines the condition under which it is called but may have nothing whatsoever to do with the nature of the raised exception
- Therefore, the use of predefined types for exceptions certainly does not promote readability
- It is much better to define classes for exceptions with meaningful names in a meaningful hierarchy that can be used for defining exceptions
- The exception parameter provides a way to pass information about an exception to the exception handler



Introduction to Event Handling

- Event handling is similar to exception handling
- In both cases, the handlers are implicitly called by the occurrence of something, either an exception or an event
- While exceptions can be created either explicitly by user code or implicitly by hardware or a software interpreter, events are created by external actions, such as user interactions through a graphical user interface (GUI)
- In conventional (non–event-driven) programming, the program code itself specifies the order in which that code is executed, although the order is usually affected by the program’s input data
- In event-driven programming, parts of the program are executed at completely unpredictable times, often triggered by user interactions with the executing program
- Particular kind of event handling discussed here is related to GUIs
- Therefore, most of the events are caused by user interactions through graphical objects or components, often called widgets
- The most common widgets are buttons
- Implementing reactions to user interactions with GUI components is the most common form of event handling

Introduction to Event Handling



- An event is a notification that something specific has occurred, such as a mouse click on a graphical button
- Strictly speaking, an event is an object that is implicitly created by the run-time system in response to a user action, at least in the context in which event handling is being discussed here
- An event handler is a segment of code that is executed in response to the appearance of an event
- Event handlers enable a program to be responsive to user actions
- Common use of event handlers is to check for simple errors and omissions in the elements of a form, either when they are changed or when the form is submitted to the Web server for processing
- Using event handling on the browser to check the validity of form data saves the time of sending that data to the server, where their correctness then must be checked by a server-resident program or script before they can be processed
- This kind of event-driven programming is often done using a client-side scripting language, such as JavaScript

Miscellaneous

Event Handling

```

import java.awt.*;
import java.awt.event.*;
class EventHandle extends Frame
    implements ActionListener
{
    TextField textField;
    EventHandle()
    {
        textField = new TextField();
        textField.setBounds(60,50,170,20);
        Button button = new Button("Quote");
        button.setBounds(90,140,75,40);
        //1
        button.addActionListener(this);
        add(button);
        add(textField);
        setSize(250,250);
        setLayout(null);
        setVisible(true);
    }
}

```

```

//2
public void actionPerformed(ActionEvent e)
{
    textField.setText("Keep Learning");
}
public static void main(String args[])
{
    new EventHandle();
}

```

