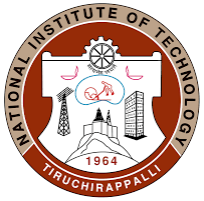


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
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	• Floating-point arithmetic computations • Arrays and Matrices • Loops and Selections	Fortran
2.	Business Applications	• Produce detailed reports • Describe and store decimal numbers and character data • Ability to specify decimal arithmetic operations	COBOL
3.	Artificial Intelligence	• Use of symbolic rather than numeric computations • Linked List of data	LISP
4.	Systems Programming	• 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	• 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

	CRITERIA		
Characteristic	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	×	-

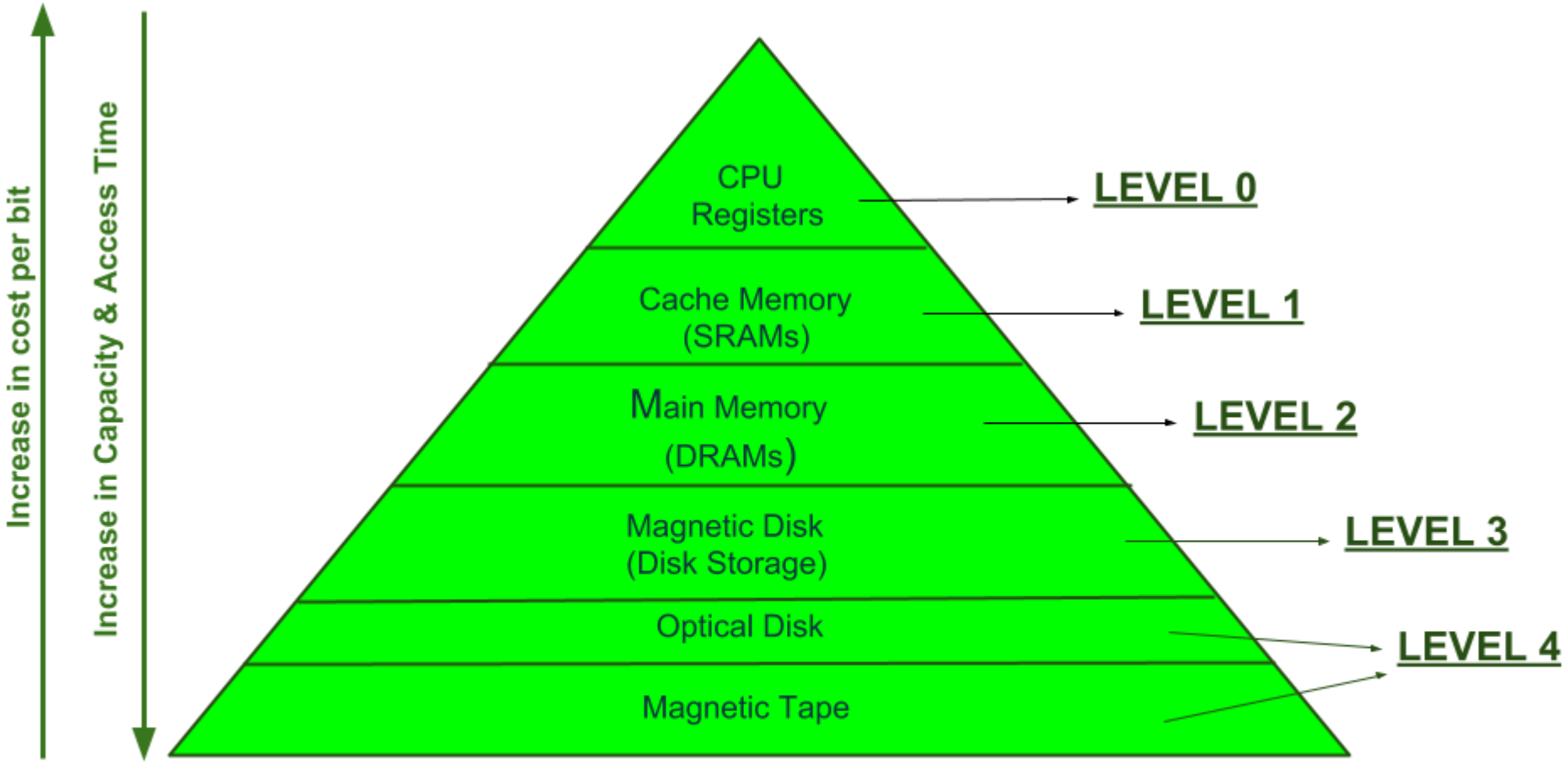
- VAX

  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 → float; b → 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)
  - $Name_1$, Age_1, $Employee\_Number_1$, $Salary_1$

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 ← $26^1 + 26^2 + 26^3 + 26^4 + 26^5 + 26^6$
  - ✓ 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

```
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 writabiltiy
  - Errors in programs can go undetected when nearly any combination of primitives is legal
  - Eg: a + b [a → float; b → int]

Support for Abstraction

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

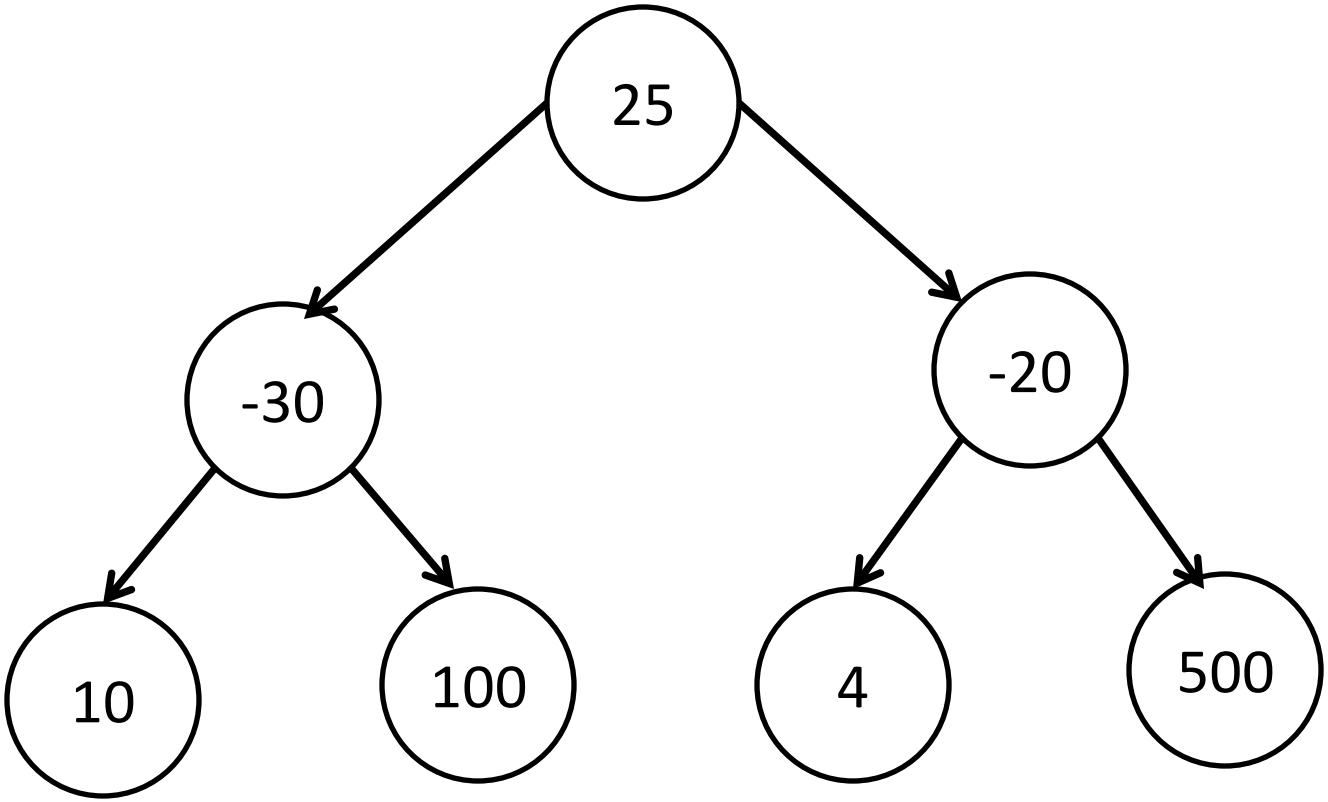
Process Abstraction

- 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	• Process Abstraction • Data Abstraction
3.	Expressivity	• 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("resut = %d", div);
    }
}
```

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

Binary Representation

Integer 23:

00000000000000000000000000010111

Float 23.0:

01000001101110000000000000000000

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

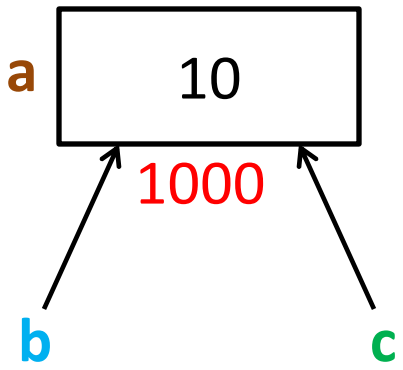
```
public class MyClass
{
   public static void main(String[ ] args)
   {
     try
     {
       int[] myNumbers = {1, 2, 3};
       System.out.println(myNumbers[10]);
     }
     catch (Exception e)
     {
       System.out.println("Something went wrong.");
     }
   }
}
```

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	• Intercept run-time errors • Take corrective action • Continue
3.	Aliasing	• 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

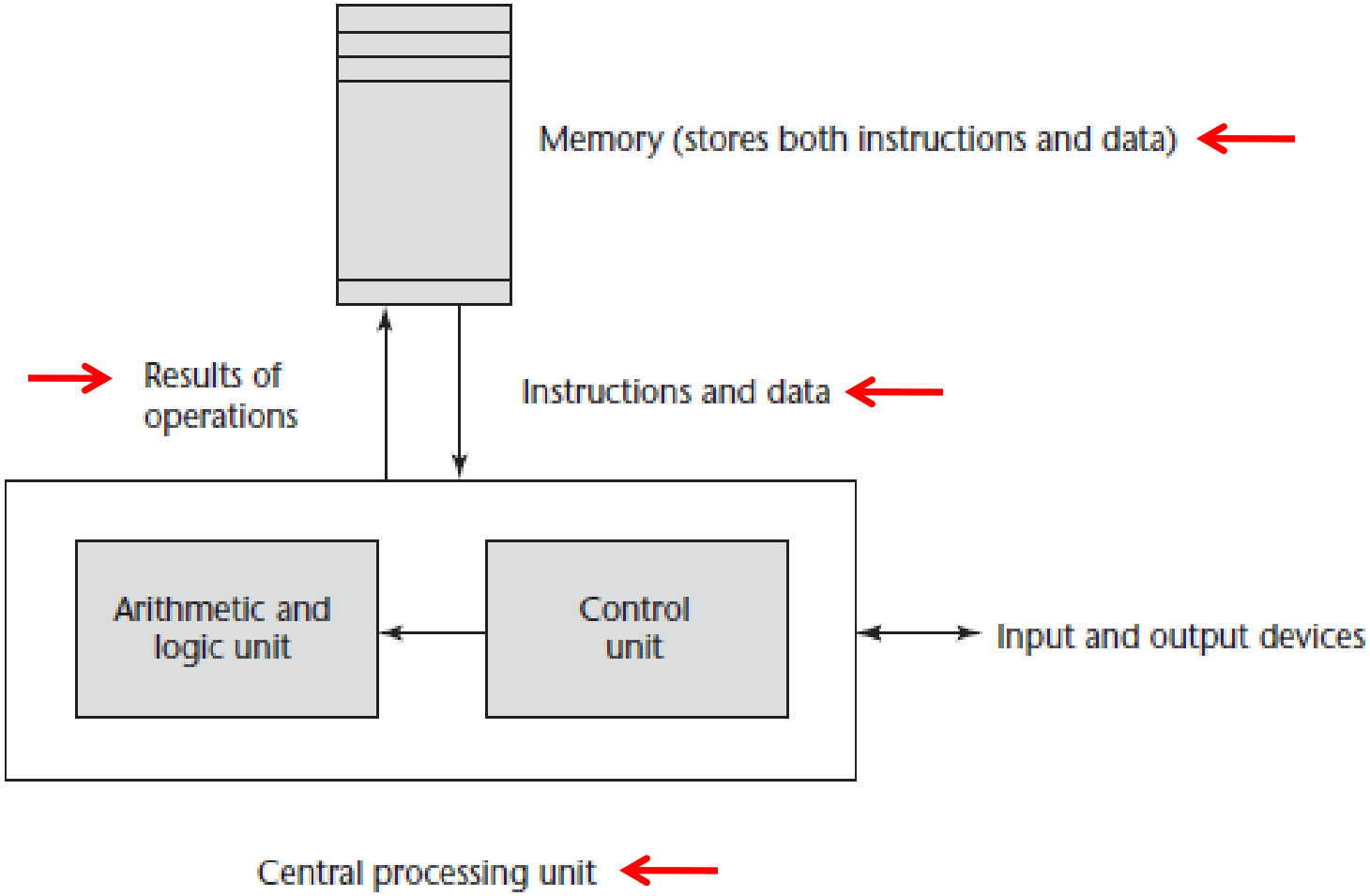
Note:

- 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 languges -> 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");
   }
   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();
   }
}
```

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

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

https://www.javatpoint.com/static-binding-and-dynamic-binding

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

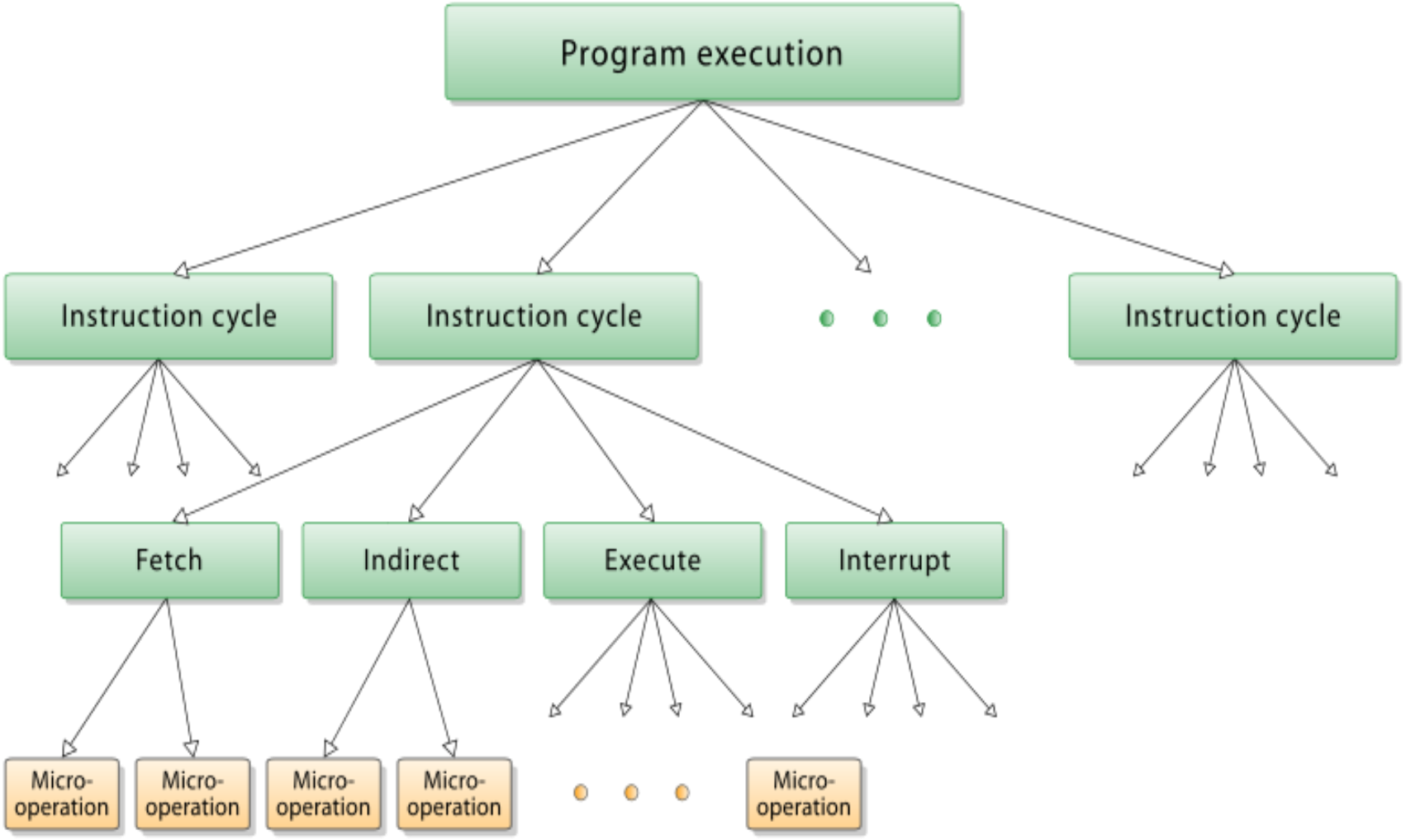
- Performs basic operations on data stored in one or more registers

  ✓ 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.

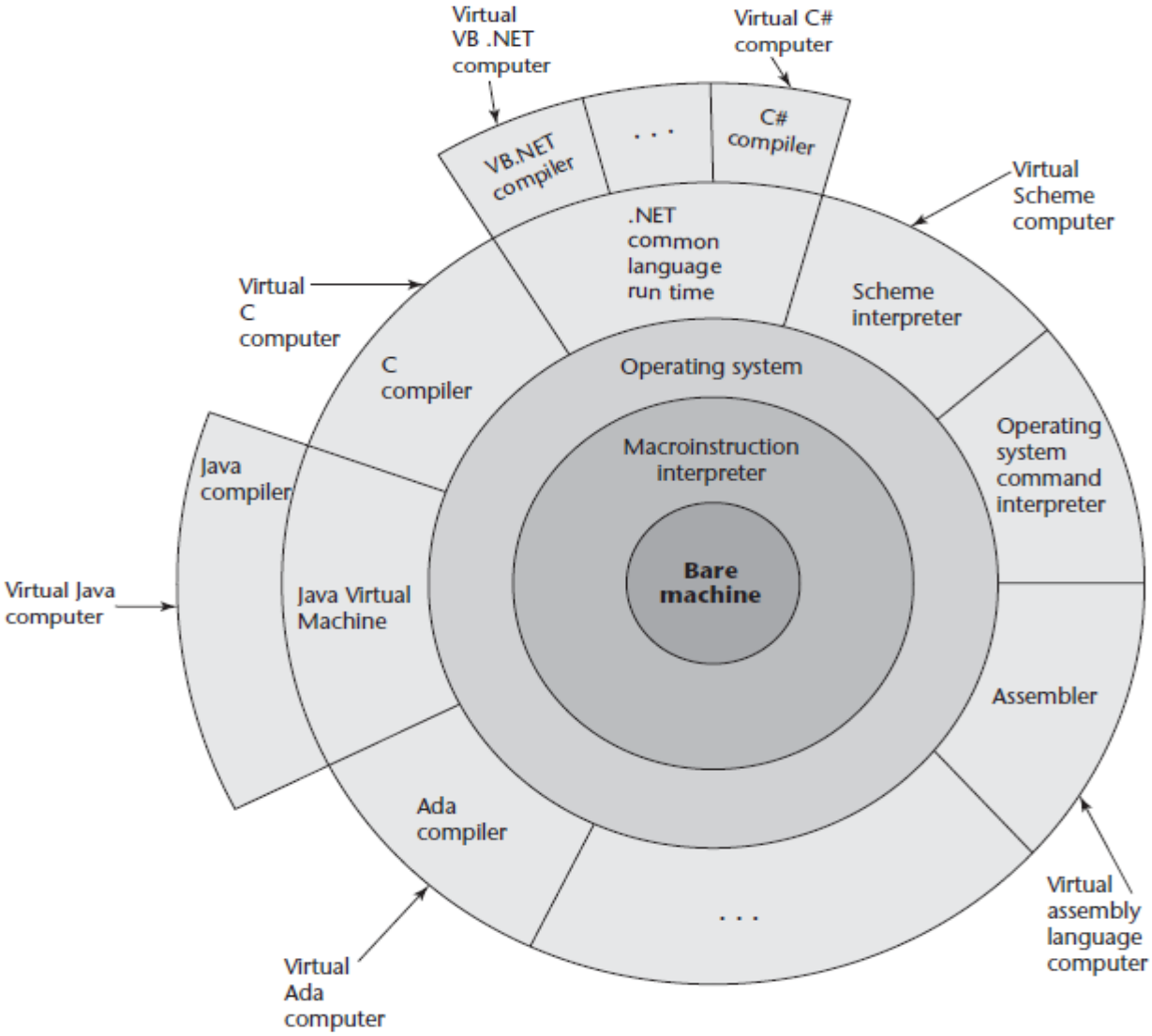
Implementation Methods

- 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

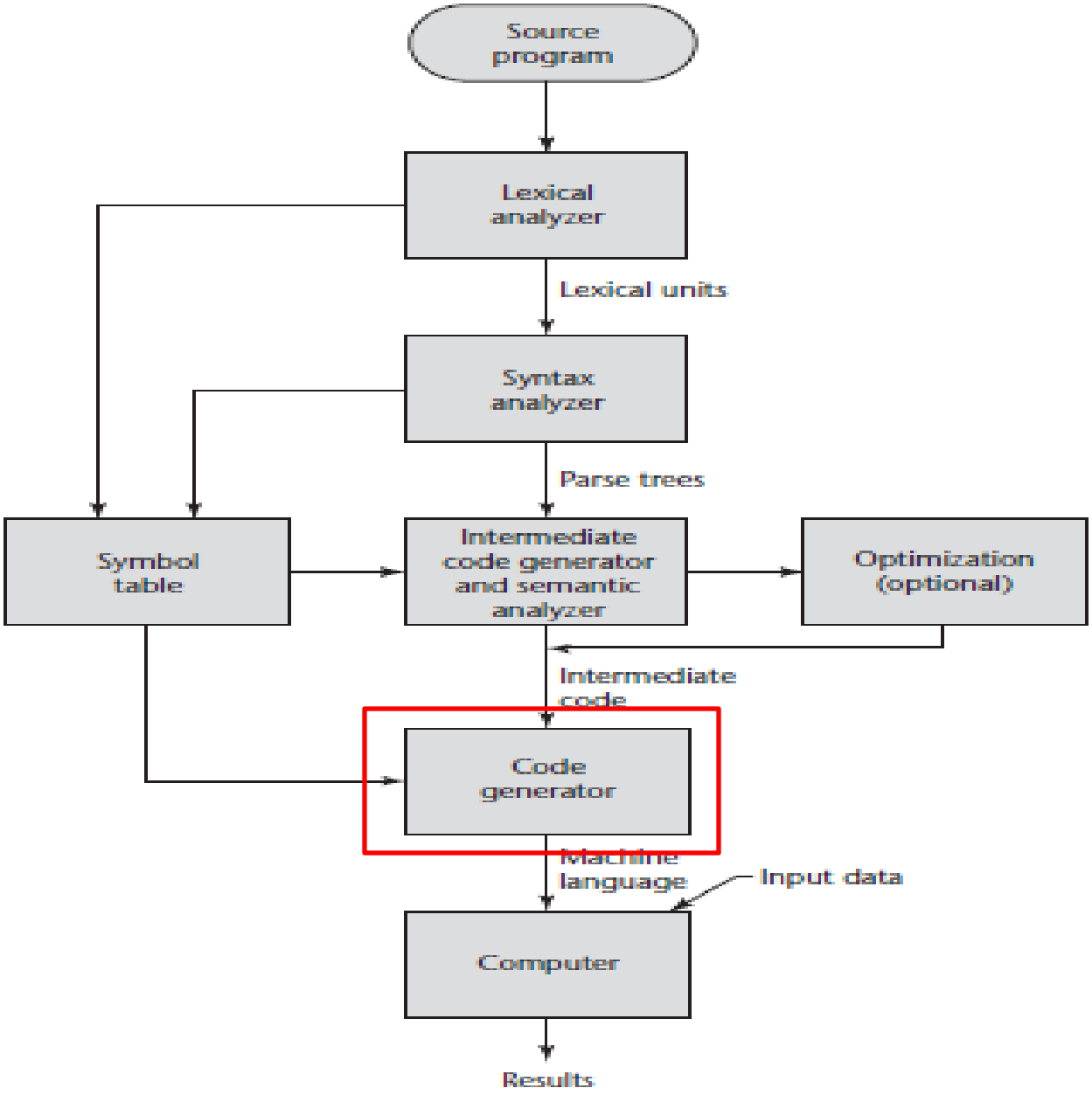
- Compiler
- Pure Interpretation
- Hybrid Implementation Systems
- Preprocessors

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

$$id_1 = id_2 + id_3 * 60$$

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

Portion = Initial + Rate * 60

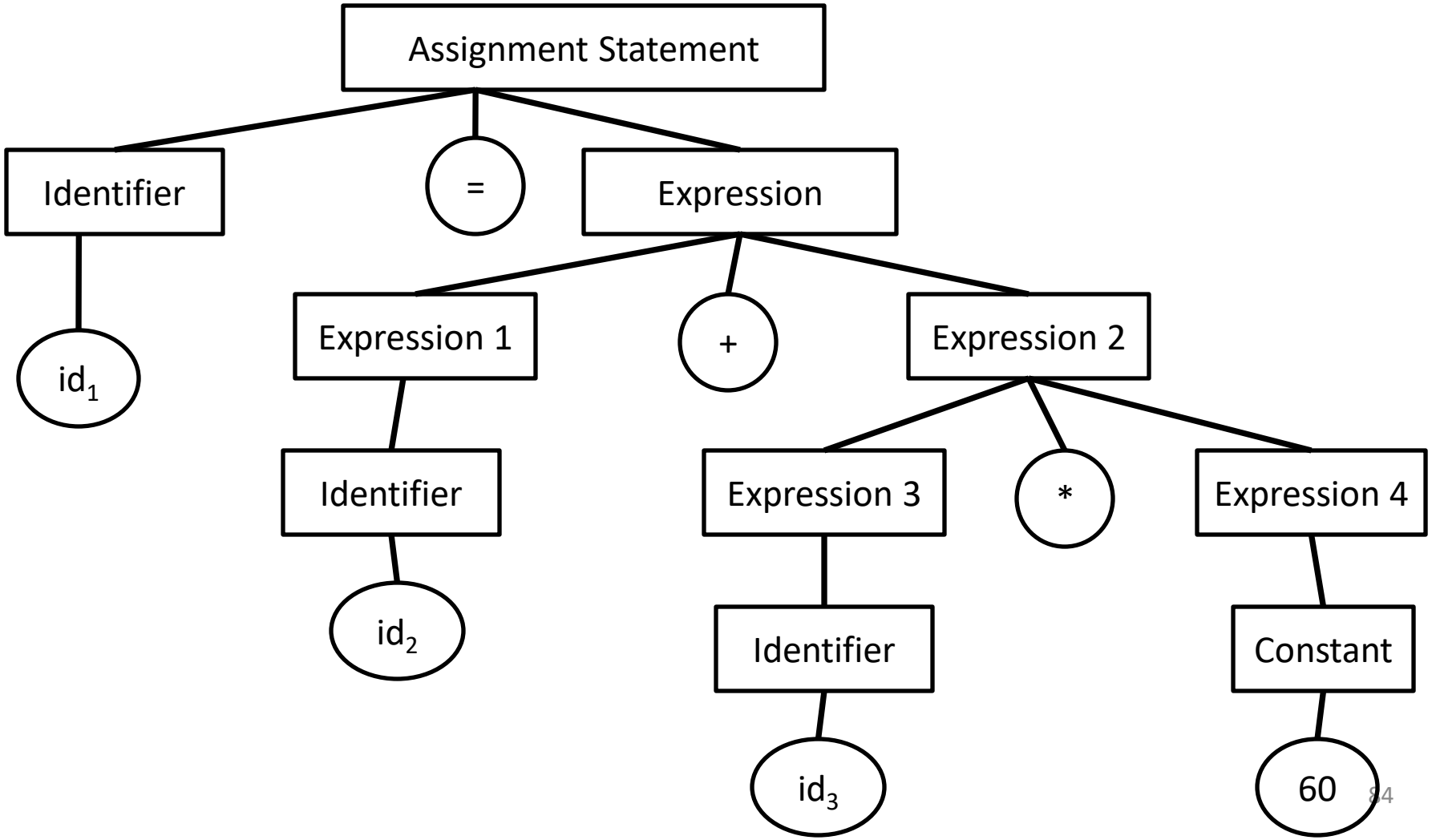
Variable Name	Data Type	Mapped Identifier$_{\#}$
Portion	int	id_1
Initial	int	id_2
Rate	int	id_3

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

$id_1 = id_2 + id_3 * 60$



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

$id_1 = id_2 + id_3 * 60$

$temp_1 = id_3 * 60$

$temp_2 = id_2 + temp_1$

$id_1 = temp_2$

General Format:

identifier = $operand_1$ operator $operand_2$

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

$$\mathbf{id_1 = id_2 + id_3 * 60}$$

$temp_1 = id_3 * 60$

$temp_2 = id_2 + temp_1$

$id_1 = temp_2$

$temp_1 = id_3 * 60$

$id_1 = id_2 + temp_1$

Code Generator

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

$$\mathbf{id_1 = id_2 + id_3 * 60}$$

$temp_1 = id_3 * 60$

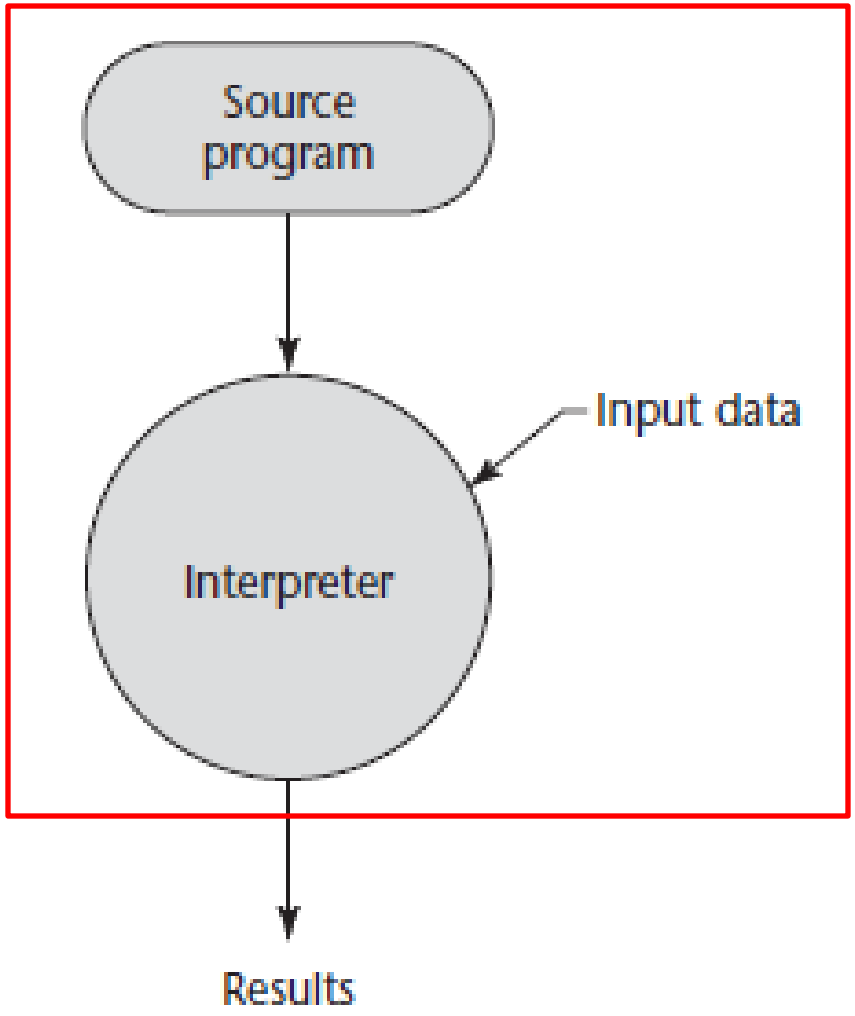
$id_1 = id_2 + temp_1$

MOV id_3, R_1
MUL #60, R_1
MOV id_2, R_2
ADD R_1, R_2
MOV R_2, 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



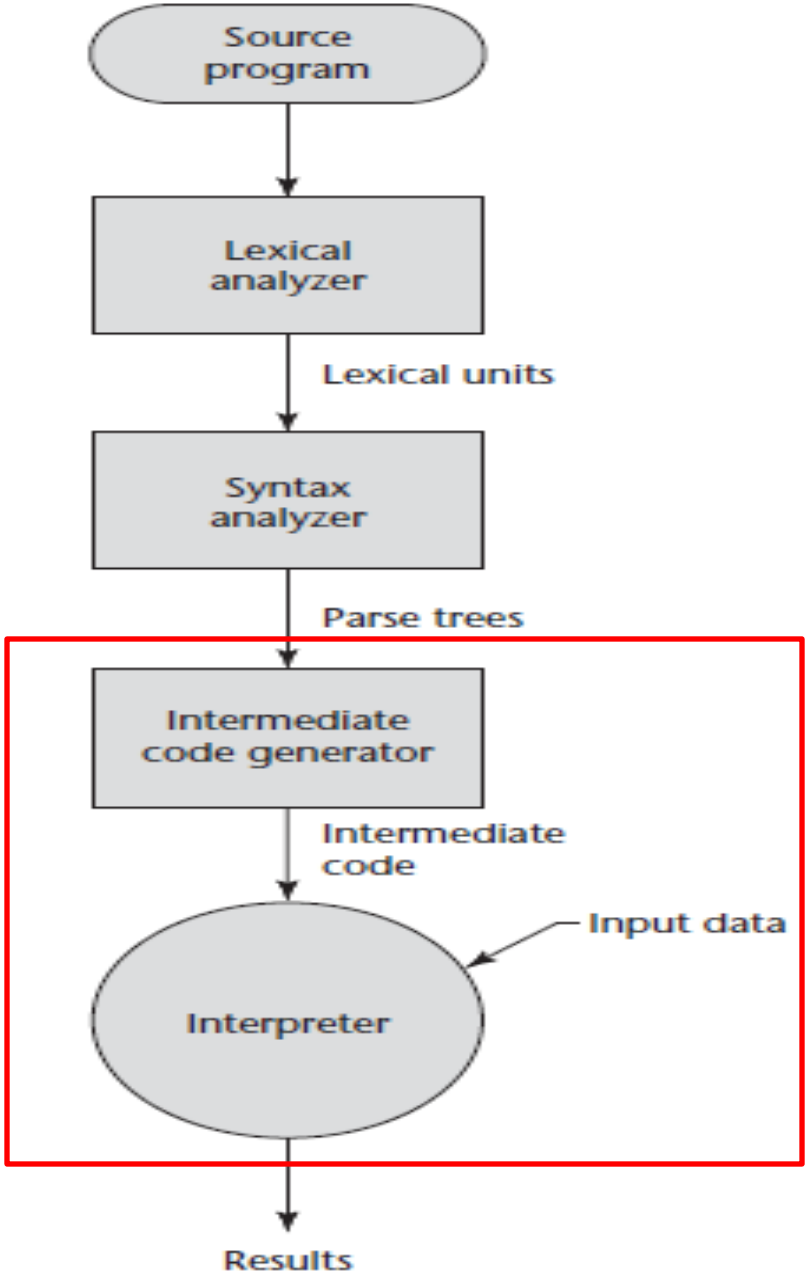
Pure Interpretation

- 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