Unit I
Introduction

By B Lokesh Joel

UNIT-1:
Title and Content Layout with List

- **The Role of programming Languages:**
  - Towards Higher-level Languages
  - Programming Paradigms
  - Criteria for good
  - language design and Language implementation.
- **Language Description :**
  - Expression notation
  - Abstract syntax tree
  - Context free Grammars

What is a Programming Language?

- a tool for instructing machines
- a means of communicating between programmers
- a vehicle for expressing high-level designs
- a notation for algorithms
- a way of expressing relationships between concepts
- a tool for experimentation
- a means for controlling computerized devices

Language Designers

- Balance
- … making computing **convenient** for *people with*
- and making **efficient** use of computing machines

Levels

- Gross distinction between programming language
- based on readability
- based on independence
- based on purpose (specific … general)

Generations of Programming Languages [Levels]

1GL: machine codes

2GL: symbolic assemblers

3GL: (machine-independent) imperative languages (FORTRAN, Pascal, C ...)

4GL: domain specific application generators

5GL: AI languages …

Each generation is at a higher level of abstraction

Machine Level [is unitelligible]

- 00000010101111001010
- 00000010101111001000
- 00000011001110101000
- Can you tell what this code fragment does?
- Can it be executed on any machine?

Assembly Language

- LD R1,"0"
- LD R2, M
- ST R2, R1
- … real assembly used mnemonics
- Add A(M), …. Had to do your own indexing
- What does this program do?

Basic Concepts of a RAM machine

- Memory: addresses, contents
- Program: instructions
- input/output:(files)

	2000		200A
A	20	c	10

A = 3 + c

Lvalue→ address

Rvalue→contents

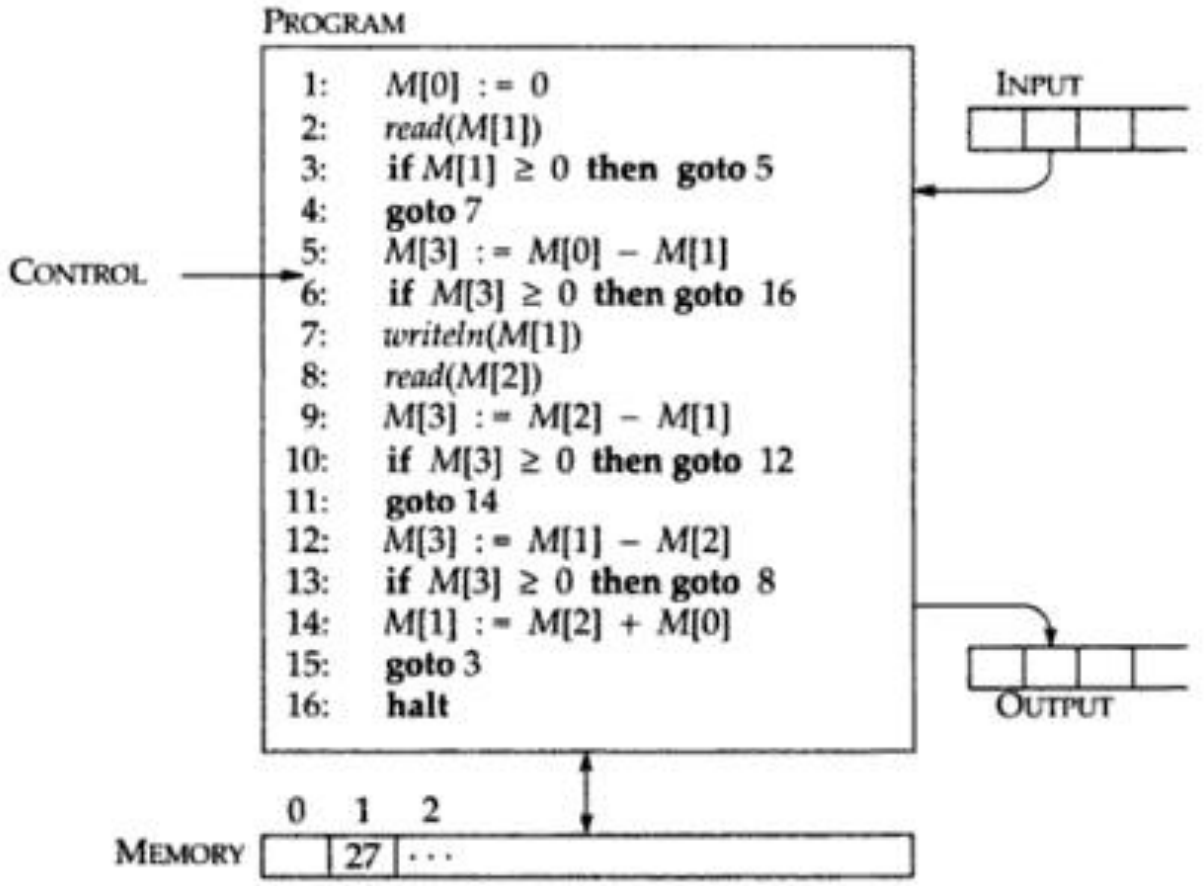


Figure 1.1 A random-access machine (RAM).

High Level [Benefits]

- Readable familiar notations
- machine independence
- availability of program libraries
- consistency check (check data types)

Problems of Scale

- Changes are easy to make
- isolated program fragments can be understood
- BUT… **one small bug can lead to disaster**
- read the NOT story about Mariner rockets
- Notice how the chairman does not understand that a "small" problem can lead to devastating result and why it was not caught

Bugs

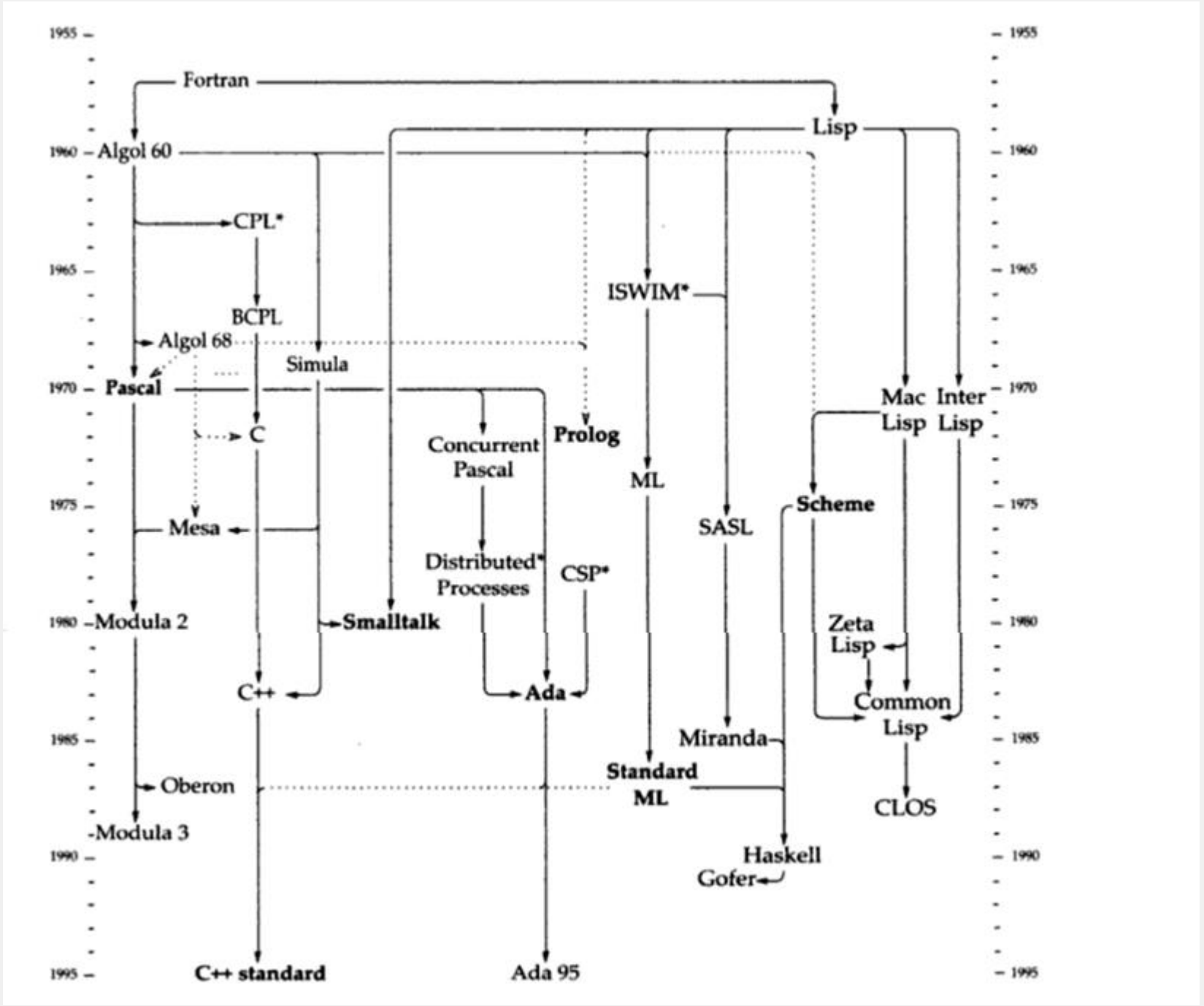
- *Programming testing can be used to show the presence of bugs, but never their absence!*
- Dijkstra
- Programming Languages can help
- readable and understandable
- organize such that parts can be understood

Role of Programming Languages

- Art (science) of programming is organizing complexity
- Must organize in such a way that our limited powers are sufficient to guarantee that the computation will establish the desired effect
- (Dijkstra - structured programming, sometimes referred to as goto-less programming)

Programming Paradigms

- **Imperative -** action oriented, sequence of actions
  - examples: C, Pascal, Basic, Fortran
- **Functional -** LISP, symbolic data processing,
- **Object-Oriented-**
  - examples: C++, Java, Smalltalk
- **Logic -** Prolog, logic reasoning

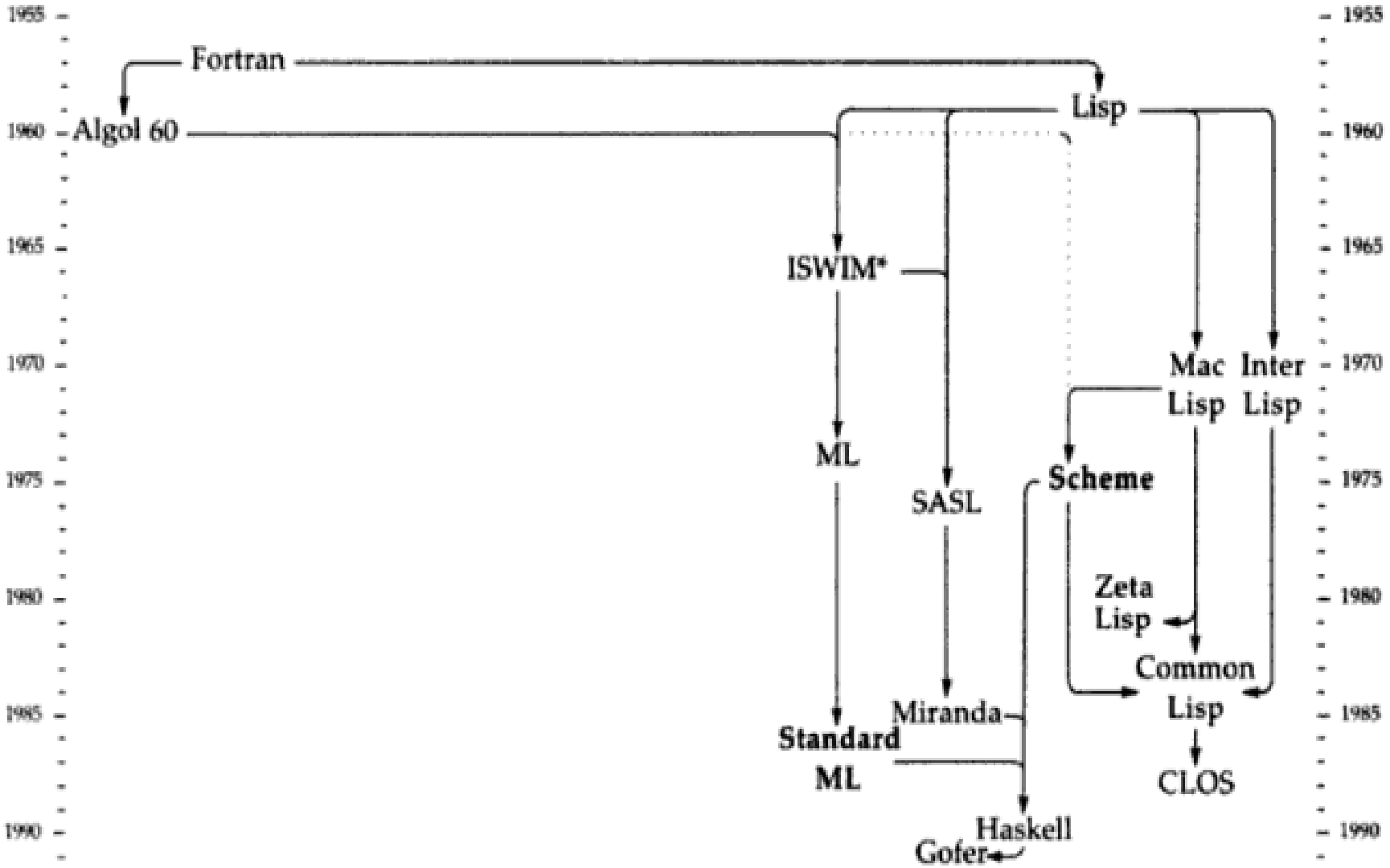

1955
1960
1965
1970
1975
1980
1985
1990
1995
Fortran
Lisp
Algol 60
CPL*
BCPL
Algol 68
Simula
Pascal
C
Concurrent Pascal
Prolog
ISWIM*
ML
SASL
Mac Lisp
Inter Lisp
Scheme
Mesa
Distributed Processes
CSP*
Modula 2
Smalltalk
Zeta Lisp
C++
Ada
Common Lisp
Miranda
Standard ML
Oberon
CLOS
Modula 3
Haskell
Gofer
C++ standard
Ada 95

Imperative[Procedural] programming

Imperative[Procedural] programming

- Fortran 1955-1960 for scientific programming
  - familiar notations
  - efficiency
- Toward common language
- Algol, Algol 60 1960s. Imperative languages is viewed as Algol family.
- Pascal was designed as a teaching language
- C 1972, created by Dennis Ritchie. C provides a rich set of operators, a terse syntax, and efficient access to the machine. C implements UNIX system.

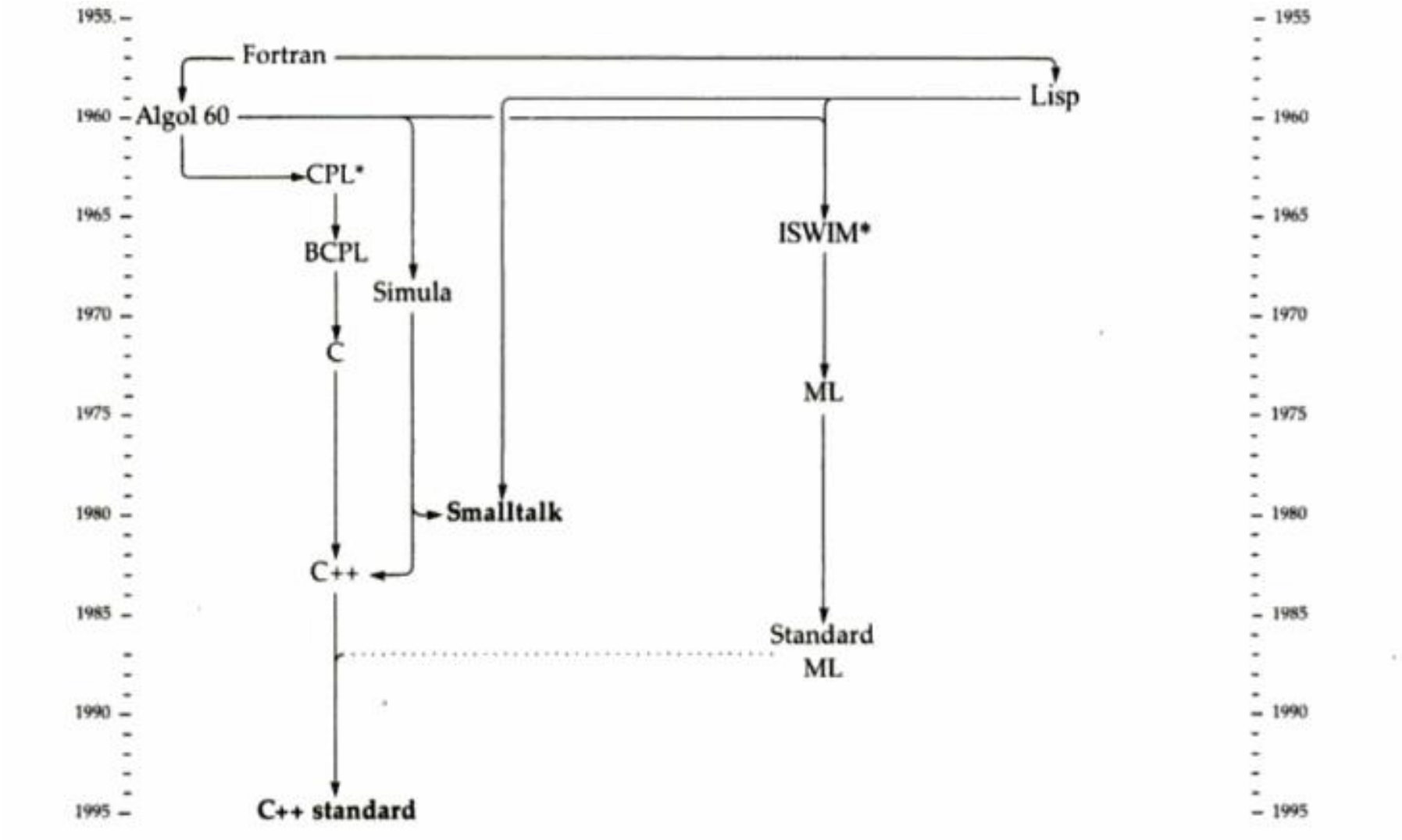
Functional programming



Functional Programming: Lisp 1958

- **Lisp** is an acronym derived from **List Processor**.
- Example:
  - (+ 2 3)
  - (Shakespeare wrote (the tempest))
  - (cdr (a b c d e))
- Program and data are both represented by lists in LISP
- Creation of common Lisp in 1984
- **CLOS** is an object-oriented extension; the full name is **Common Lisp Object System**

Object-oriented programming



Objected-Oriented Programming (1961-1967)

- From **Simula's** origins in simulation
- Key concepts from Simula is that of class of object
  - A traffic simulation can have many cars and many trucks
  - The simulation has
    - objects of the class of cars and
    - objects of the class of trucks.
  - All of which share certain properties
    - Car and trucks are all vehicles
  - The classification of objects into classes and subclasses is central to object oriented programming

Objected-Oriented Programming (1961-1967)

- Smalltalk is an interactive system with graphic user interface
- C++ was designed to bring the benefits of objects to imperative programming in C.
- Java (Network Language)
  - Embedded in your Browser
    - Platform-independent capability
    - Using virtual machine

Programming Paradigms

A programming language is a problem-solving tool.

Imperative style:	program = algorithms + data *good for decomposition*
Functional style:	program = functions ^{o} functions *good for reasoning*
Logic programming style:	program = facts + rules *good for searching*
Object-oriented style:	program = objects + messages *good for modeling(!)*

Other styles and paradigms: blackboard, pipes and filters, constraints, lists, ...

Procedural vs Functional

- Program: a sequence of instructions for a von Neumann m/c.
- Computation by instruction execution.
- Iteration.
- Modifiable or updateable variables.

- Program: a collection of function definitions (m/c independent).
- Computation by term rewriting.
- Recursion.
- *Assign-only-once* variables.

Functional Style : Illustration

- Definition : Equations

  sum(0) = 0

  sum(n) = n + sum(n-1)

- Computation : Substituition and Replacement

  sum(2)

  = 2 + sum (2-1)

  = …

  = 3

Paradigm vs Language

- Imperative Style

```
i := 0;   sum := 0;
while (i < n) do
        i := i + 1;
    sum := sum + i
end;
```

  - Storage efficient

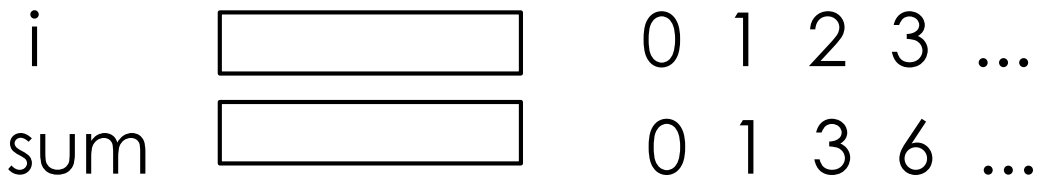
- Functional Style

```
func sum(i:int) : int;
  if i = 0
  then  0
  else i + sum(i-1)
end;
```

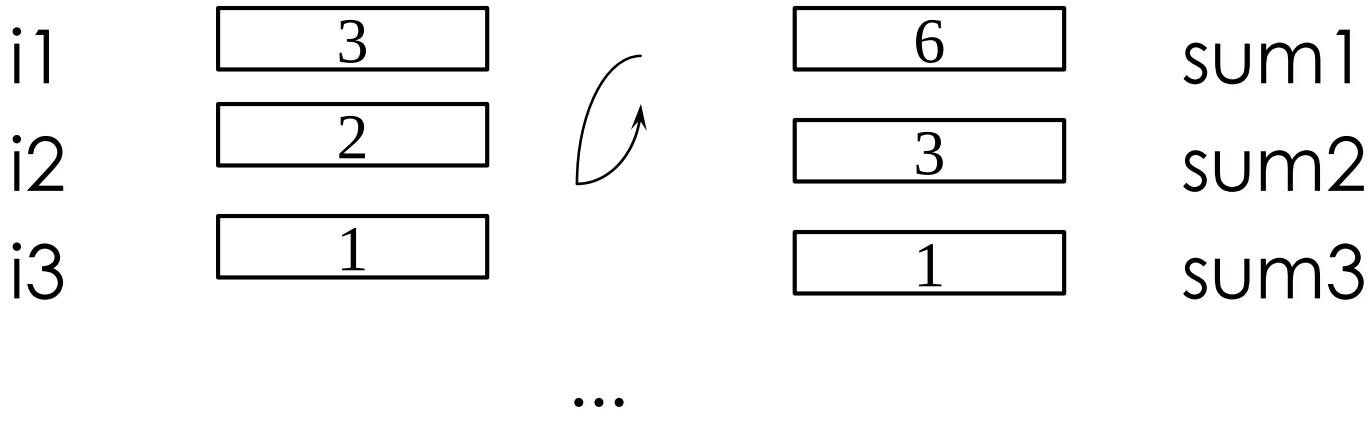
  - No Side-effect

Role of Variables

- **Imperative (read/write)**



- **Functional (read only)**



Bridging the Gap

- Tail recursive programs can be automatically optimized for space by translating them into equivalent while-loops.

```
func sum(i : int, r : int) : int;
        if  i = 0   then   r
        else   sum(i-1, n+r)
end
```

  - Scheme does not have loops.

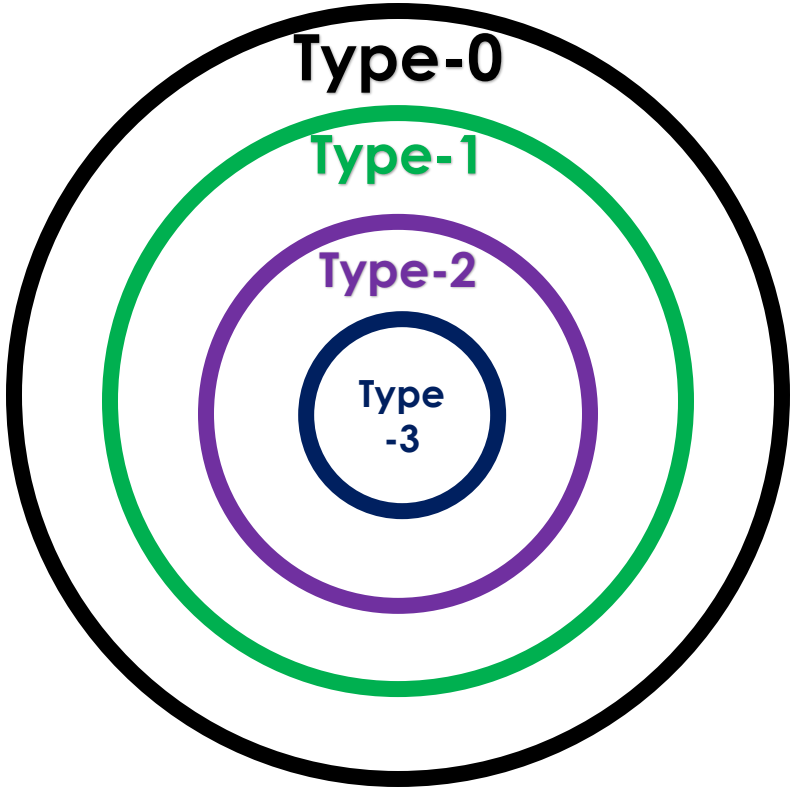
Analogy: Styles *vs* Formalisms

- Iteration
- Tail-Recursion
- General Recursion
- Regular Expression
- Regular Grammar
- Context-free Grammar

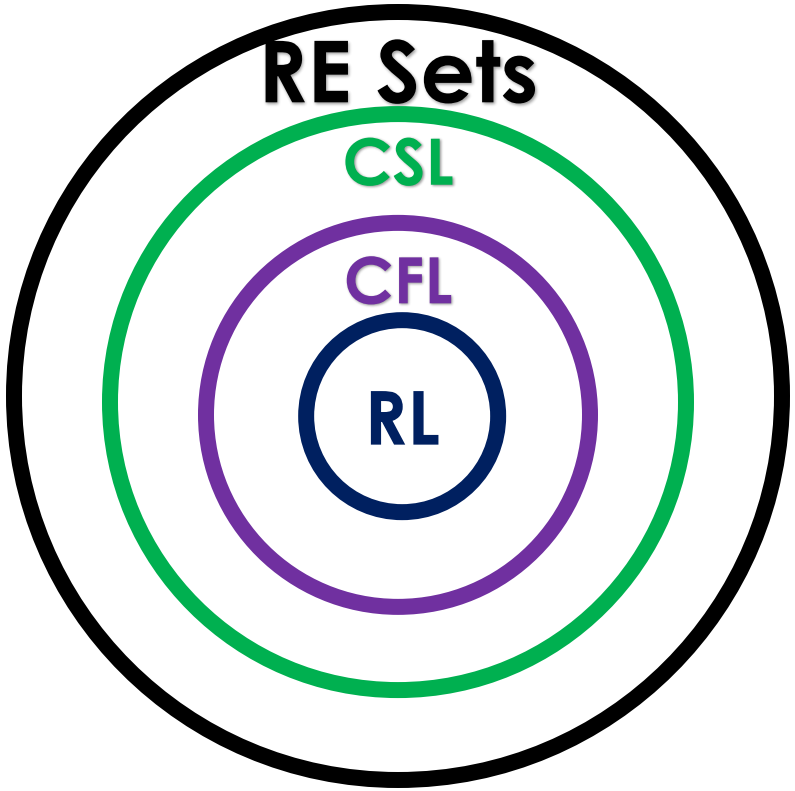
CHOMSKY HIERARCHY

CHOMSKY HIERARCHY

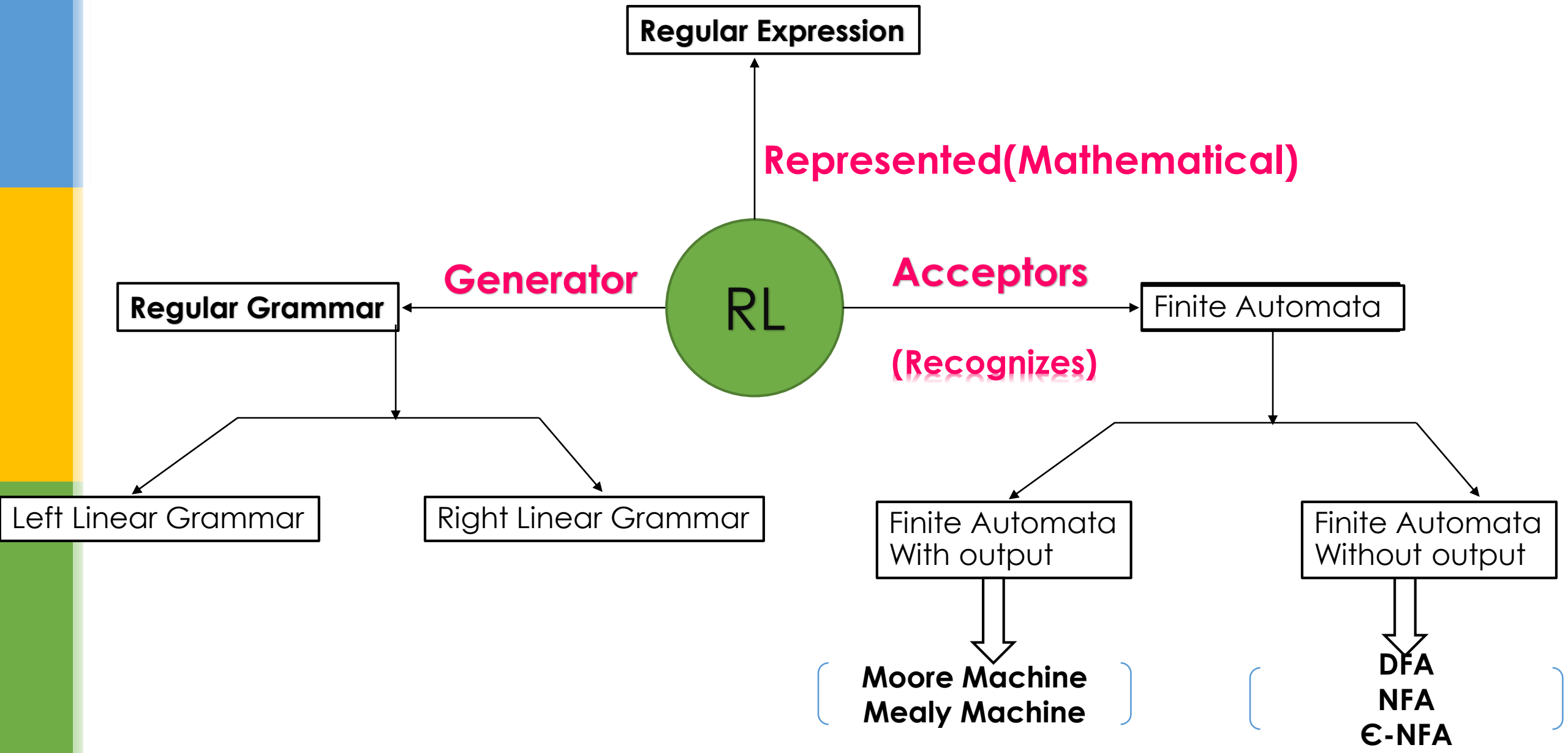
- Four Language
  1. Type 0➔ Unrestricted Lang[or Recursive Enumerable Lang]
  2. Type 1➔ Context sensitive Lang
  3. Type 2➔ Context free Lang
  4. Type 3➔ Regular Lang


Type-0
Type-1
Type-2
Type
-3

The types of languages form a strict hierarchy:



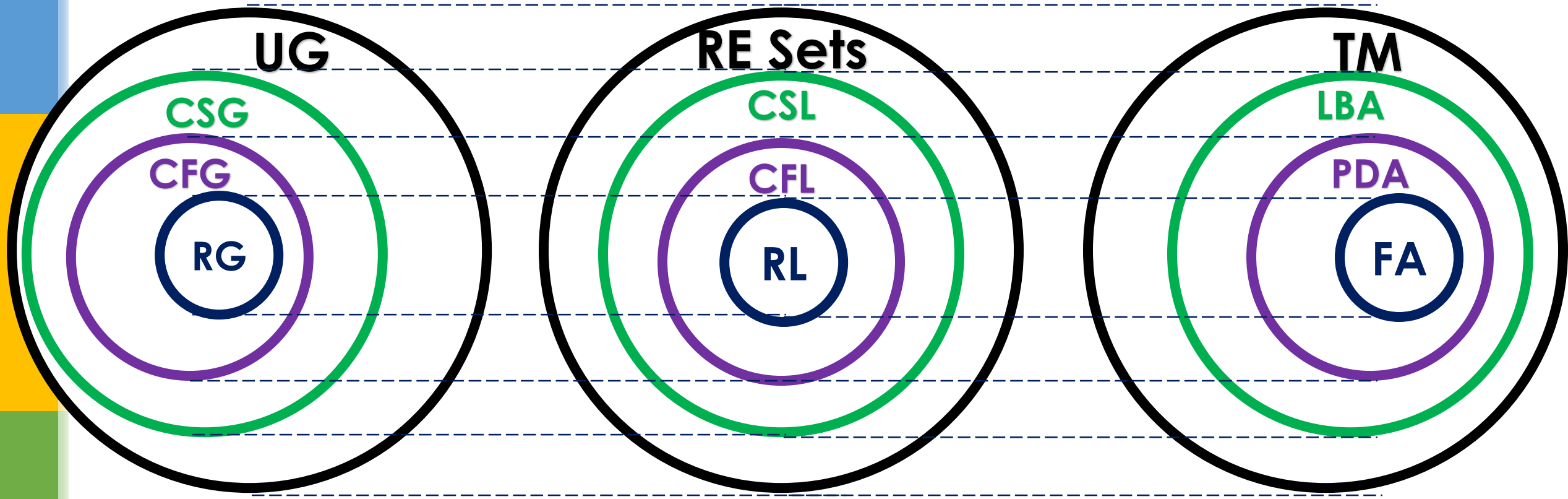
regular languages ⊂ context-free languages ⊂ context-sensitive languages ⊂ recursive languages ⊂ recursively enumerable languages.


Regular Expression
Represented(Mathematical)
Generator
RL
Acceptors
(Recognizes)
Regular Grammar
Finite Automata
Left Linear Grammar
Right Linear Grammar
Finite Automata
With output
Finite Automata
Without output
Moore Machine
Mealy Machine
DFA
NFA
Є-NFA

Grammar

Language

Machine



Object-Oriented Style

- **Programming with *Abstract Data Types***
  - ADTs specify/describe behaviors.
- **Basic Program Unit: *Class***
  - Implementation of an ADT.
    - Abstraction enforced by encapsulation.
- **Basic Run-time Unit: *Object***
  - Instance of a class.
    - Has an associated *state*.

Procedural vs Object-Oriented

- Emphasis on procedural abstraction.
- Top-down design;

  Step-wise refinement.
- Suited for programming in the small.

- Emphasis on data abstraction.
- Bottom-up design;

  Reusable libraries.
- Suited for programming in the large.

Integrating Heterogeneous Data

- **In C, Pascal, etc., use**

  Union Type / Switch Statement

  Variant Record Type / Case Statement

- **In C++, Java, Eiffel, etc., use**

  Abstract Classes / Virtual Functions

  Interfaces and Classes / Dynamic Binding

Comparison : *Figures* example

- Data
  - Square
    - side
  - Circle
    - radius
- Operation (area)
  - Square
    - side * side
  - Circle
    - PI * radius * radius

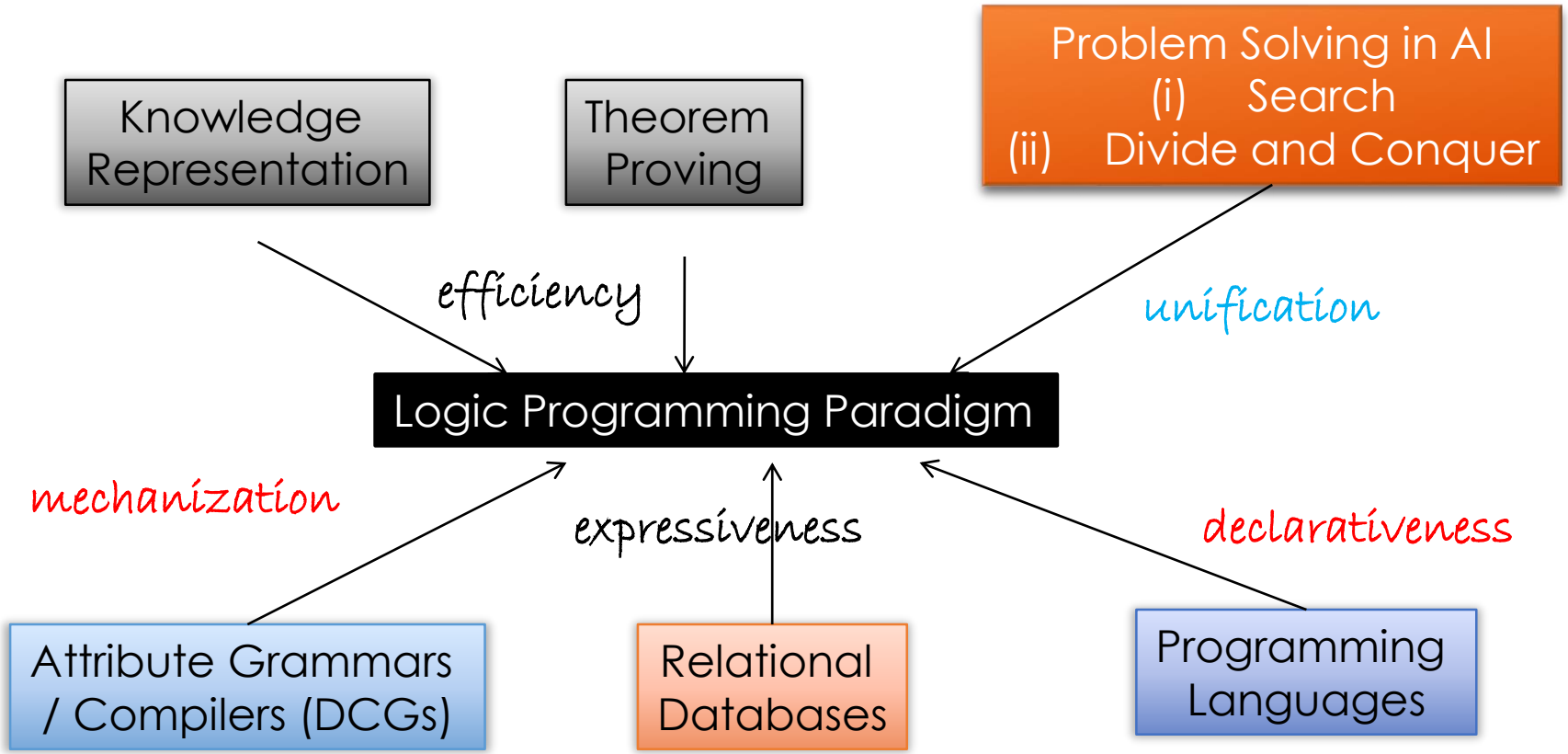
- Classes
  - Square
    - side
    - area
      (= side * side)
  - Circle
    - radius
    - area
      (= PI*radius*radius)

Logic Programming Paradigm

- Integrates Data and Control Structures

```
edge(a,b).
edge(a,c).
edge(c,a).
path(X,X).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),  path(Z,Y).
```

Trading expressiveness for efficiency : Executable specification



Language Implementation: Bridging the Gap

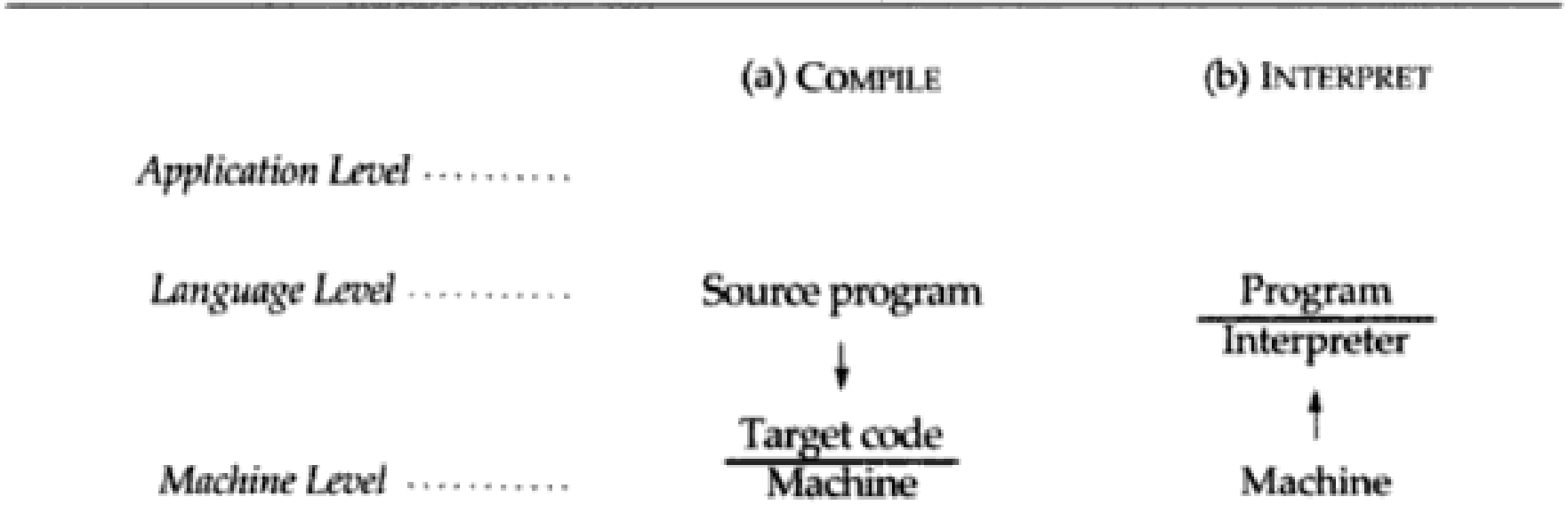
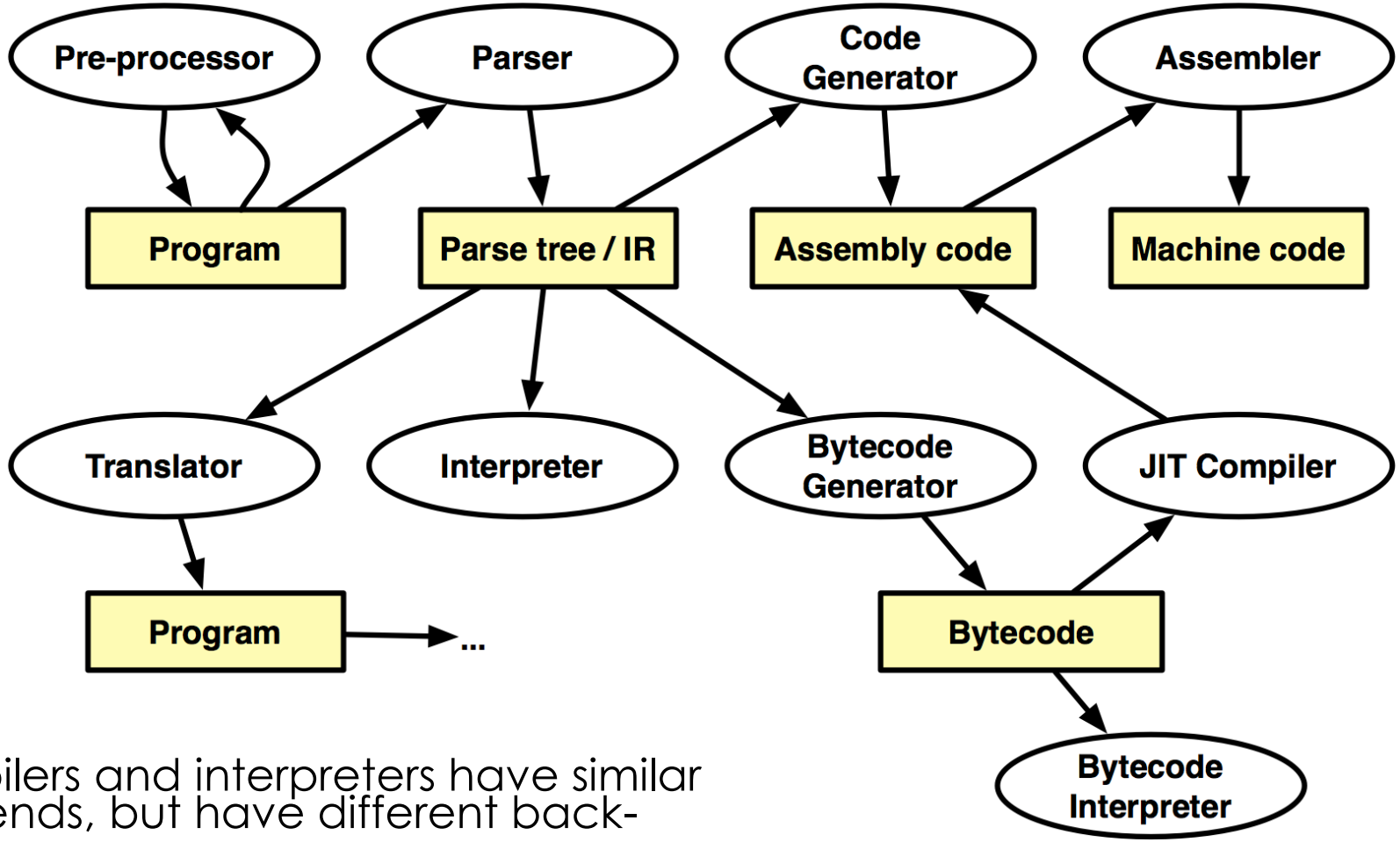


Figure 1.7 A language can be implemented by (a) translating it down to the level of the machine, or (b) by building a higher-level machine called an interpreter that can run the language directly.

Compilers and Interpreters



Compilers and interpreters have similar front-ends, but have different back-ends.

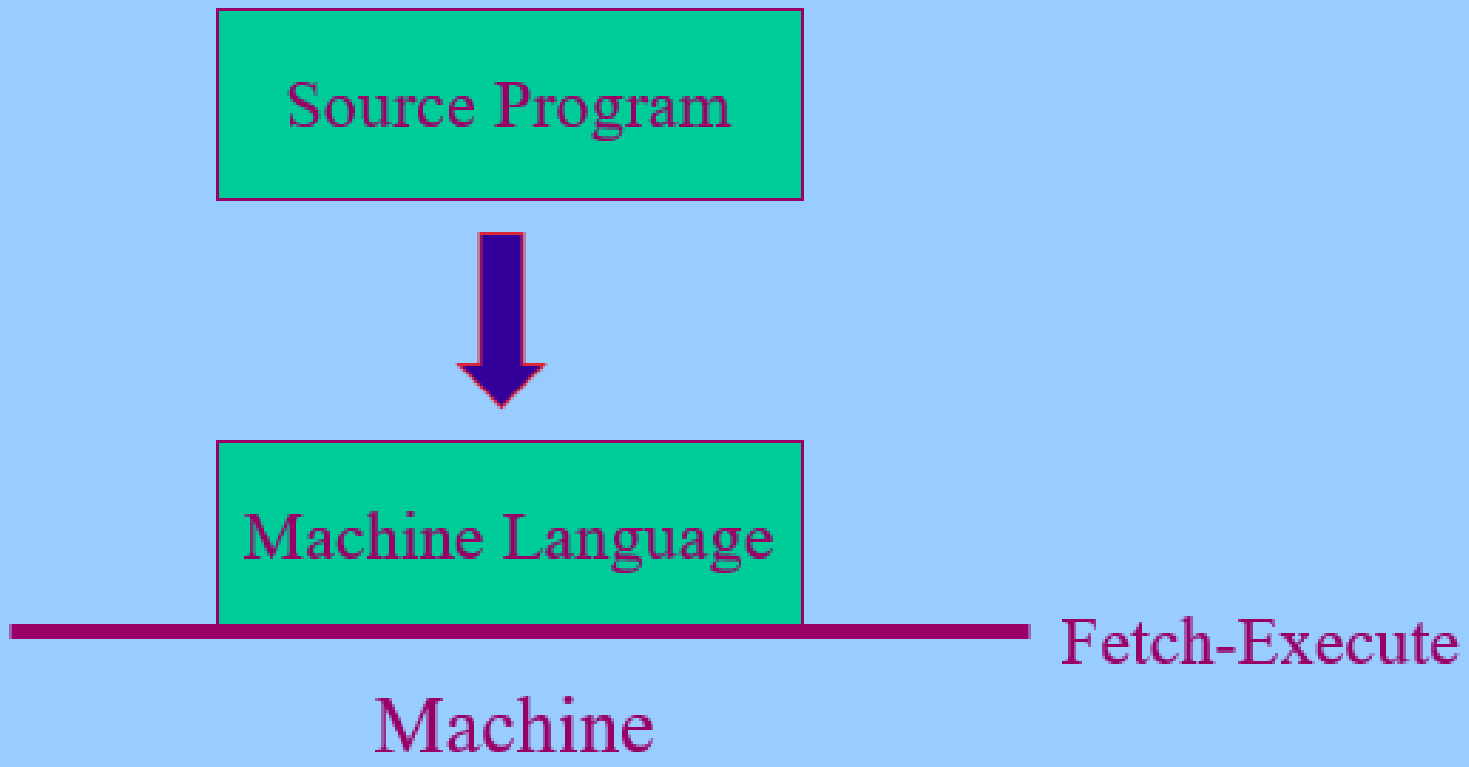
Language Implementation

- **Compiler -** source code it translated into machine code (all at once)
- **Interpreter -** machine is brought up to the language (one statement at a time)

Compiler

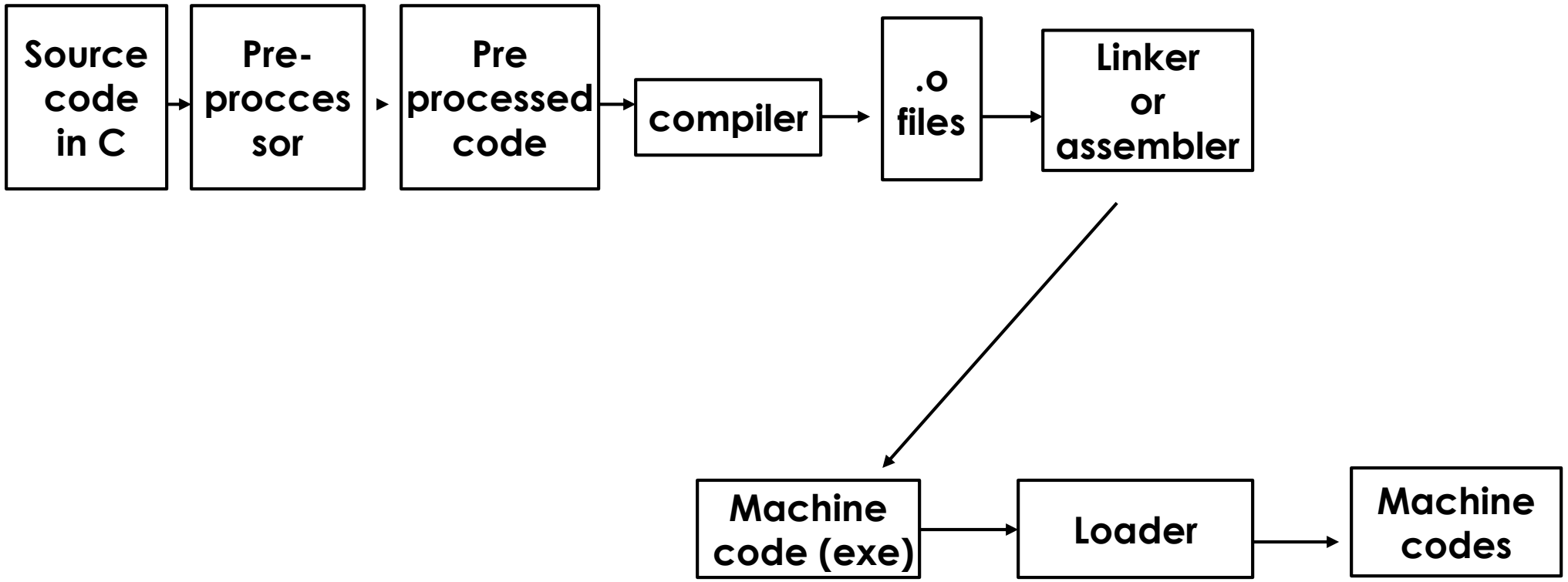
- Compiler translates source program into an equivalent program in the target language.
- Once the translation is complete, the target code is run at a later time, called run time.
- Compilation can be more efficient than interpreter

Implementation Methods 1. Compilation

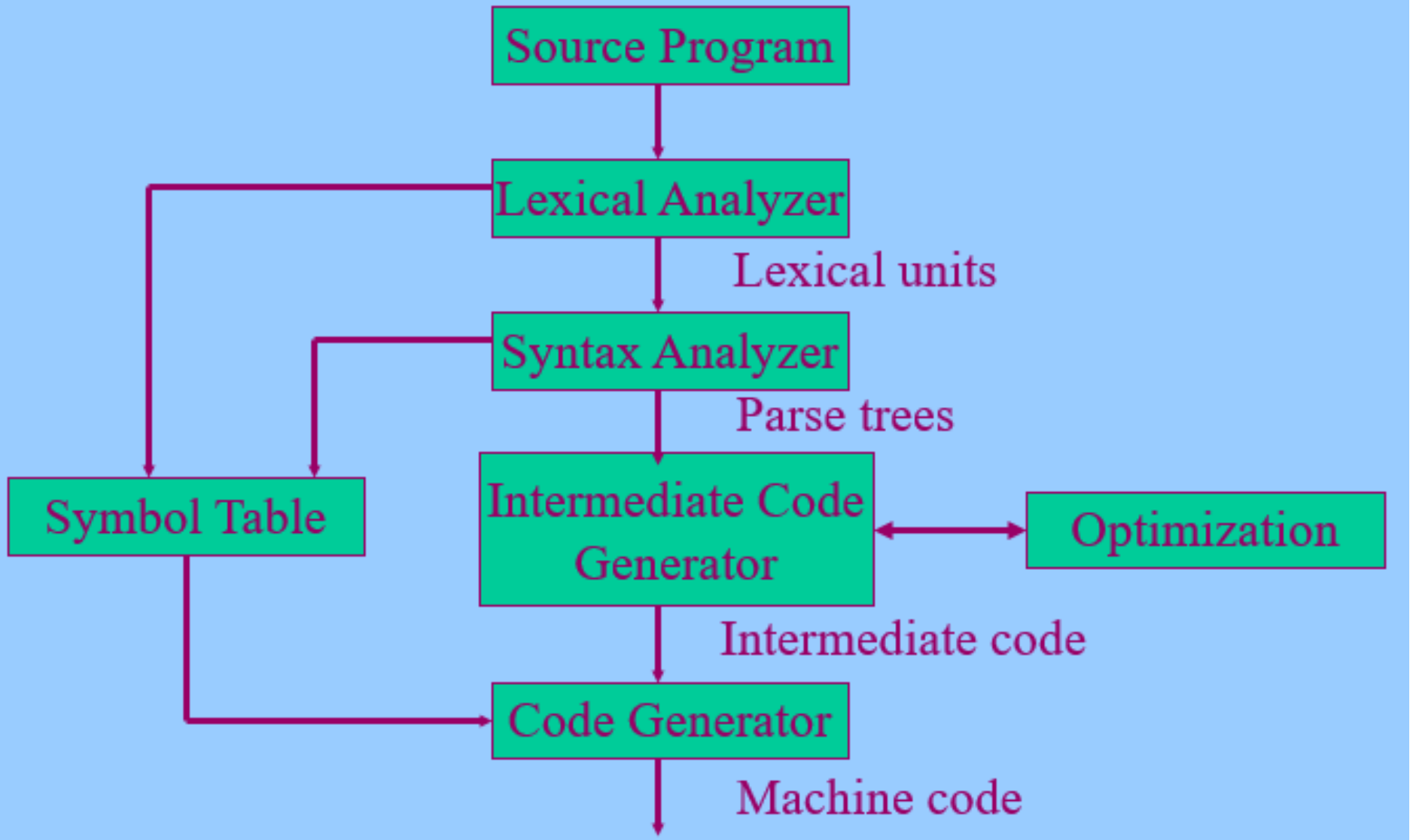


- Source programs are translated to machine language

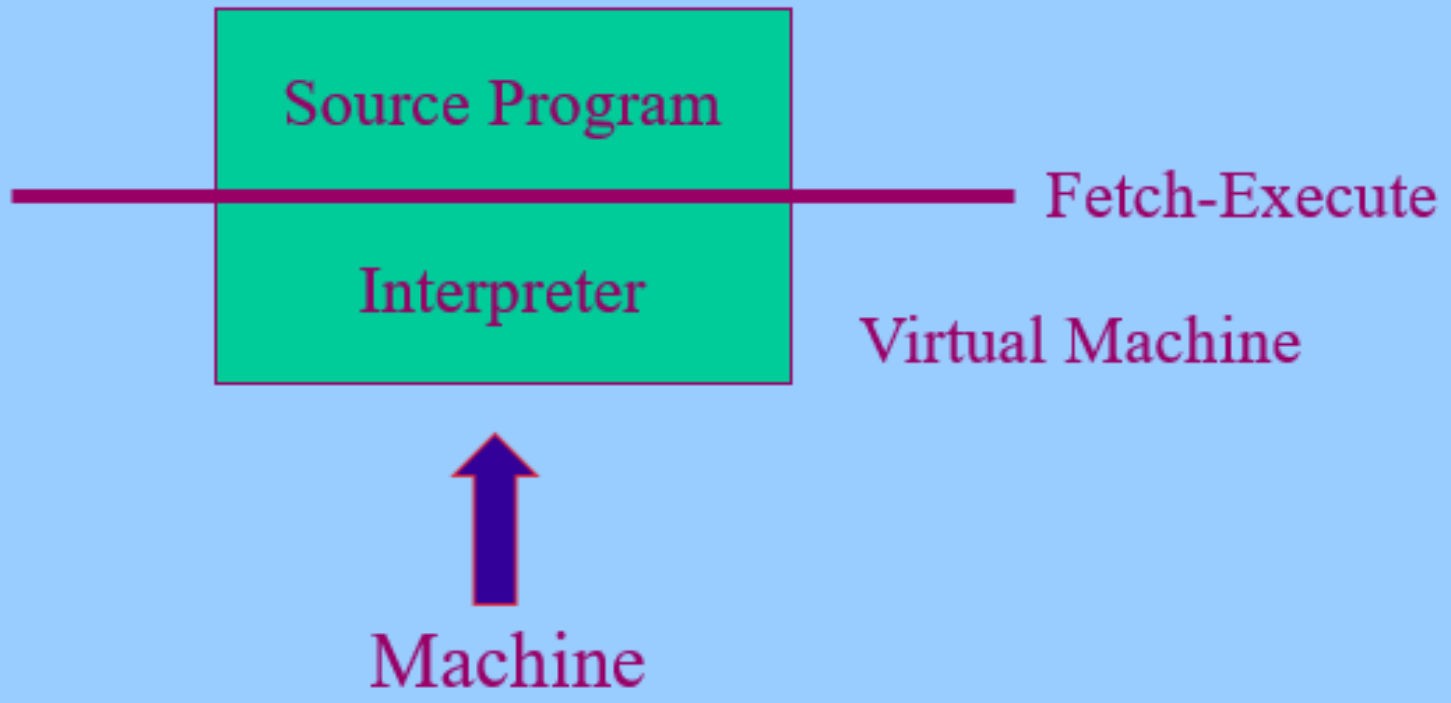
Compiled C


Source code in C
Pre-procces sor
Pre processed code
compiler
.o files
Linker or assembler
Machine code (exe)
Loader
Machine codes

Implementation Methods 1. Compilation



Implementation Methods 2. Pure Interpretation



- Source programs are directly interpreted by another program without any translation.

Interpreted Code

- It takes a program and its input at the same time
- It scans the program, implementing operations as it encounters them, and doing input/output as needed.
- Interpreter can be more flexible than compilation.
- Each instruction is interpreted by machine interpreter
- does not produce object code

Comparisons

- Compilation more efficient
- Interpreted more flexible

Part 2