CS49K Programming Languages

Chapter 1: Introduction

LECTURE 1: INTRODUCTION

DR. ERIC CHOU

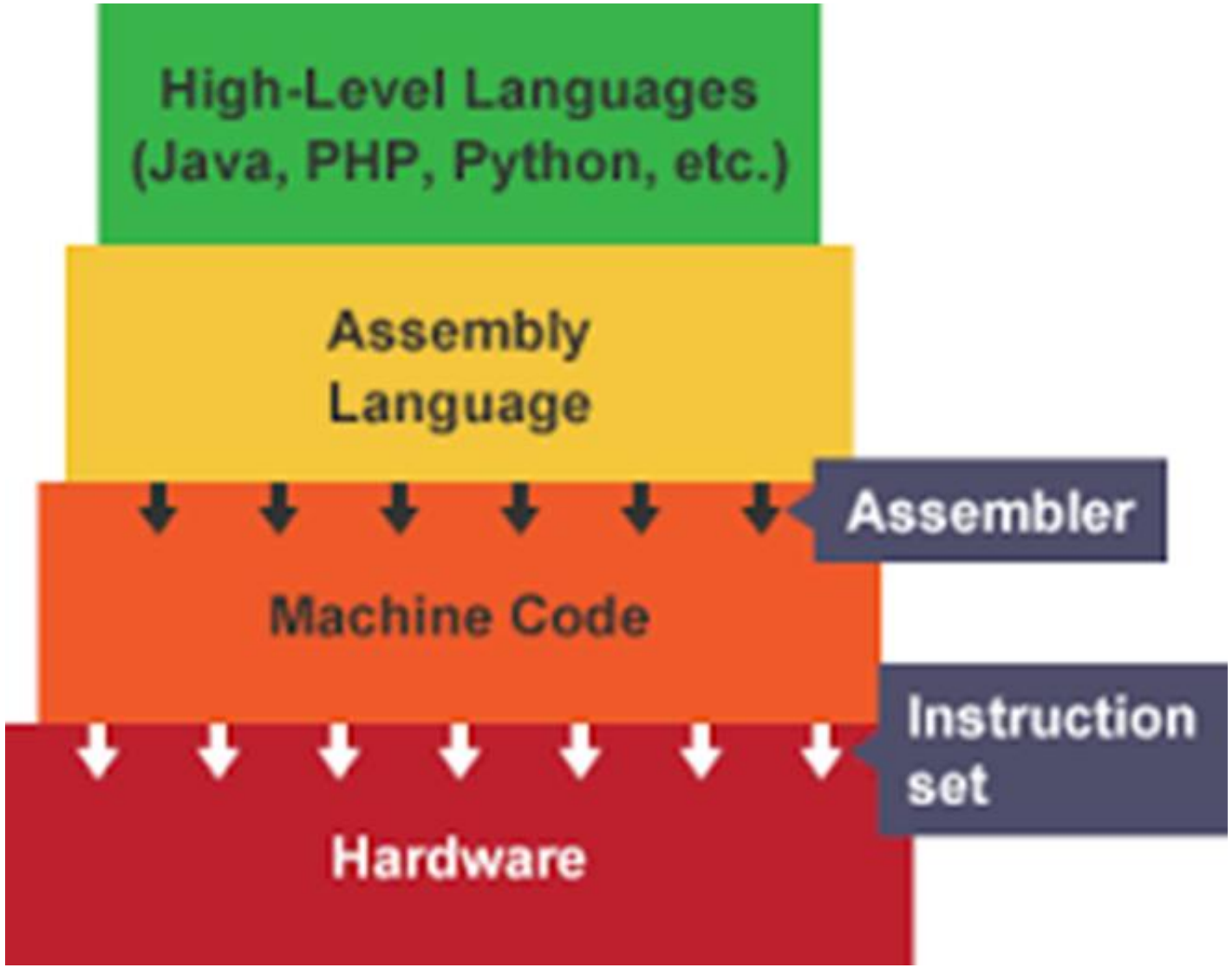
IEEE SENIOR MEMBER

Objectives

- Introduction to Programming Languages
- The history of programming languages
- Translation Levels: Compilation and Interpretation
- Scripting Languages versus programming languages

Introduction to Programming Languages

SECTION 1


High-Level Languages
(Java, PHP, Python, etc.)
Assembly
Language
Assembler
Machine Code
Instruction
set
Hardware

Machine Code

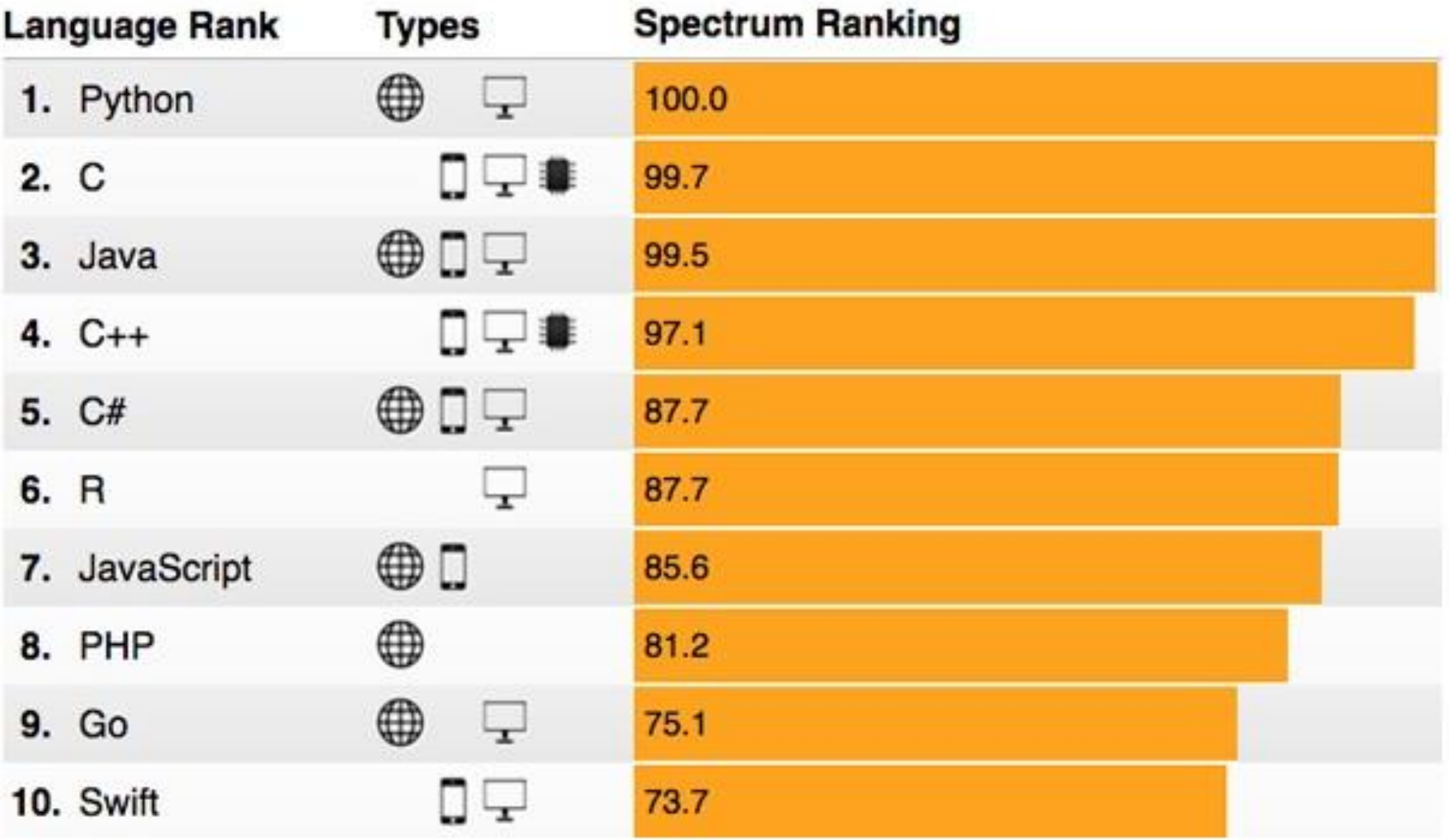
Unreadable

```
    pushl   %ebp              # \
    movl    %esp, %ebp        #  ) reserve space for local variables
    subl    $16, %esp         # /
    call    getint            # read
    movl    %eax, -8(%ebp)    # store i
    call    getint            # read
    movl    %eax, -12(%ebp)   # store j
A:  movl    -8(%ebp), %edi    # load i
    movl    -12(%ebp), %ebx   # load j
    cmpl    %ebx, %edi        # compare
    je      D                 # jump if i == j
    movl    -8(%ebp), %edi    # load i
    movl    -12(%ebp), %ebx   # load j
    cmpl    %ebx, %edi        # compare
    jle     B                 # jump if i < j
    movl    -8(%ebp), %edi    # load i
    movl    -12(%ebp), %ebx   # load j
    subl    %ebx, %edi        # i = i - j
    movl    %edi, -8(%ebp)    # store i
    jmp     C
B:  movl    -12(%ebp), %edi   # load j
    movl    -8(%ebp), %ebx    # load i
    subl    %ebx, %edi        # j = j - i
    movl    %edi, -12(%ebp)   # store j
C:  jmp     A
D:  movl    -8(%ebp), %ebx    # load i
    push    %ebx              # push i (pass to putint)
    call    putint            # write
    addl    $4, %esp          # pop i
    leave                     # deallocate space for local variables
    mov     $0, %eax          # exit status for program
    ret                       # return to operating system
```

Assembly Language

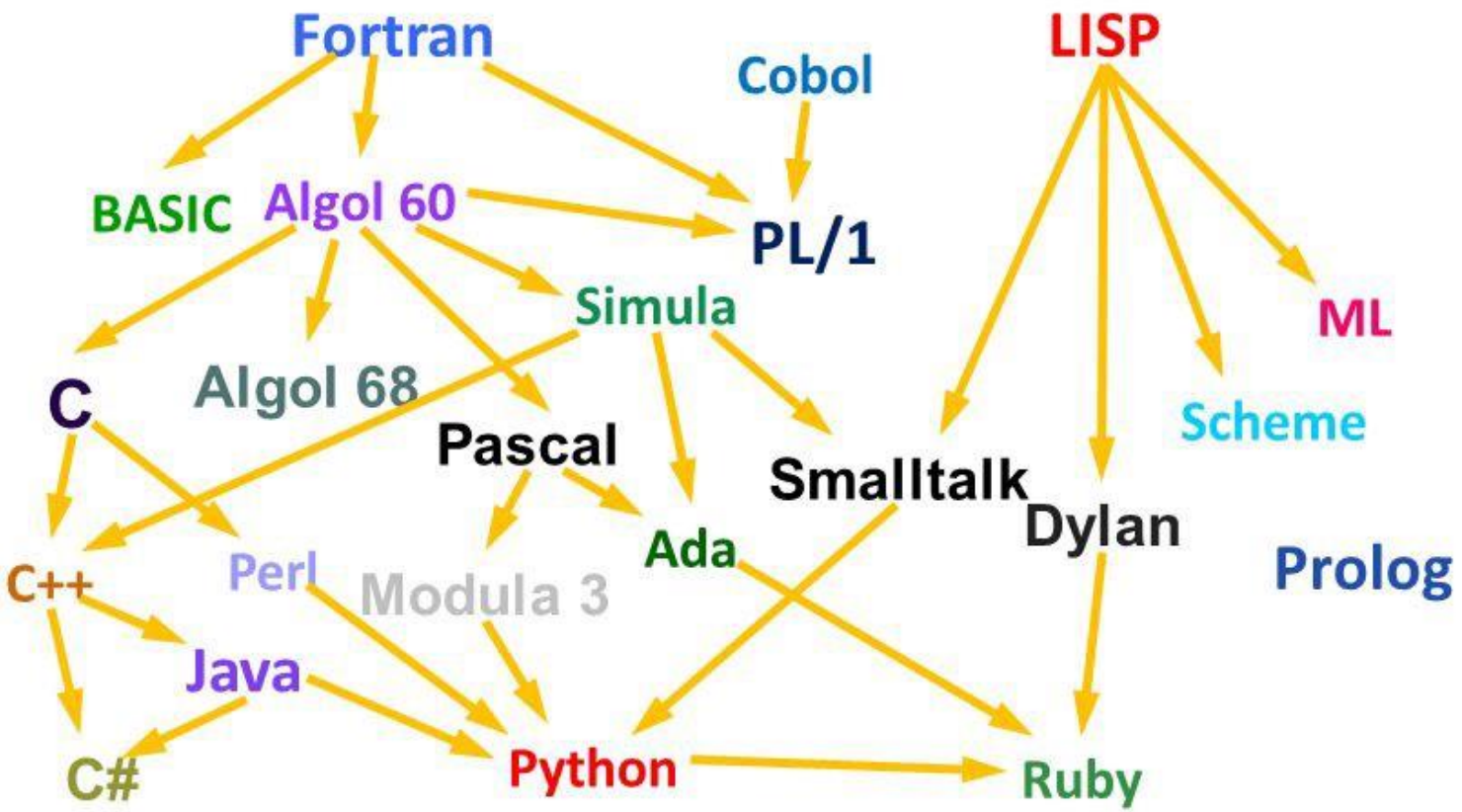
GCD Code

Most Popular Programming Languages 2021


Language Rank
Types
Spectrum Ranking
1. Python
100.0
2. C
99.7
3. Java
99.5
4. C++
97.1
5. C#
87.7
6. R
87.7
7. JavaScript
85.6
8. PHP
81.2
9. Go
75.1
10. Swift
73.7

A family tree of languages

Some of the 2400 + programming languages



Why are there so many programming languages?

1. evolution -- we've learned better ways of doing things over time
2. socio-economic factors: proprietary interests, commercial advantage
3. orientation toward special purposes
4. orientation toward special hardware
5. http://cdn.oreillystatic.com/news/graphics/prog_lang_poster.pdf

What makes a language successful?

1. easy to learn (BASIC, Pascal, LOGO, Scheme)
2. easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
3. easy to implement (BASIC, Forth, Python)
4. possible to compile to very good (fast/small) code (Fortran, C)
5. backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic, C#)
6. wide dissemination at minimal cost (Pascal, Turing, Java, Javascript)

Why do we have programming languages? What is a language for?

1. way of thinking -- way of expressing algorithms
2. languages from the user's point of view
3. abstraction of virtual machine -- way of specifying what you want
4. the hardware to do without getting down into the bits
5. languages from the implementor's point of view

The Art of Language Design

- Evolution
- Special Purposes
- Personal Preference
- Expressive Power
- Ease of Use for Novice
- Ease of Implementation
- Standardization
- Open Source
- Excellent Compiler
- Economics/Patronage/Inertia

Why Study Programming Languages?

SECTION 2

Help you choose a language

1. C vs. Modula-3 vs. C++ for systems programming
2. Python vs. Fortran vs. APL vs. Ada for numerical computations
3. C/C++ vs. Ada vs. Modula-2 for embedded systems
4. Common Lisp vs. Scheme vs. ML for symbolic data manipulation
5. Java vs. C#(.Net) for networked PC programs

Network (Web-server) Languages

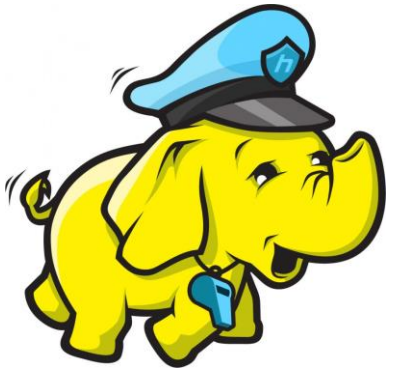
Mobile (App) Languages

Desktop (.exe) Languages

Markup/Data Languages

Database Languages

Number and Data Processing Languages



Hardware Description Languages

Electronics Languages

Make it easier to learn new languages

Some languages are similar; easy to walk down family tree

concepts have even more similarity; if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum.

Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European). East Asian Languages need to pick up Kanji Characters (CJKV, Sino-Tibetan and Altaic Languages)

Help you make better use of whatever language you use (I)

- understand obscure features:
  - In C, help you understand unions, arrays & pointers, separate compilation, varargs, catch and throw
  - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals

Help you make better use of whatever language you use (II)

- understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
  - use simple arithmetic equal (use x*x instead of x**2)
  - use C pointers or Pascal "with" statement to factor address calculations
  - avoid call by value with large data items in Pascal
  - avoid the use of call by name in Algol 60
  - choose between computation and table lookup (e.g. for cardinality operator in C or C++)

Help you make better use of whatever language you use (III)

- figure out how to do things in languages that don't support them explicitly:
  - lack of suitable control structures in Fortran
  - use comments and programmer discipline for control structures
  - lack of recursion in Fortran, CSP, etc
  - write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)

Help you make better use of whatever language you use (IV)

- figure out how to do things in languages that don't support them explicitly:
  - lack of named constants and enumerations in Fortran
  - use variables that are initialized once, then never changed
  - lack of modules in C and Pascal use comments and programmer discipline
  - lack of iterators in just about everything fake them with (member?) functions

Study of Programming Languages

- What is available?
- What is not available?
- What is good?
- What is bad?
- What is the use?

The programming Language Spectrum

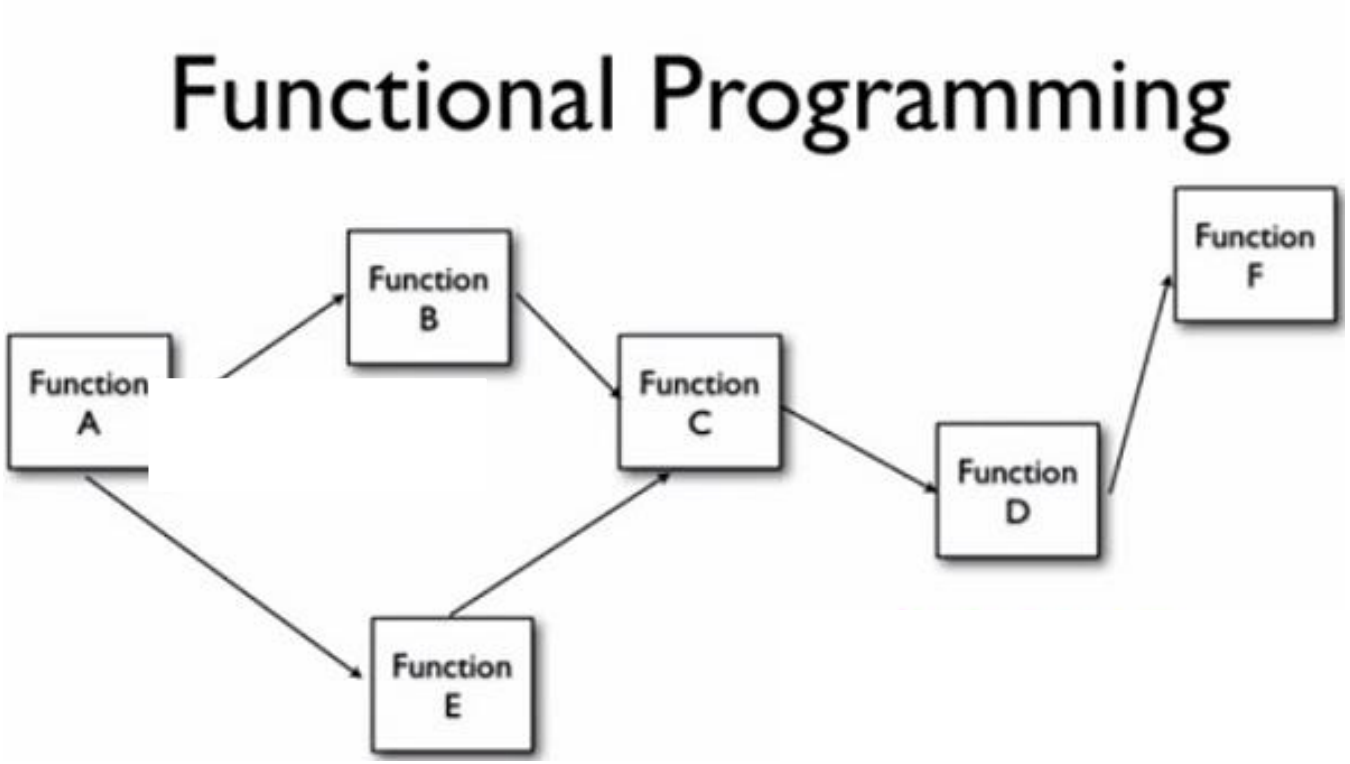
SECTION 3

Computer Scientist Group Language as …

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
imperative	
von Neumann	C, Ada, Fortran, …
object-oriented	Smalltalk, Eiffel, Java, …
scripting	Perl, Python, PHP, …

Programming Paradigm

Declarative Languages

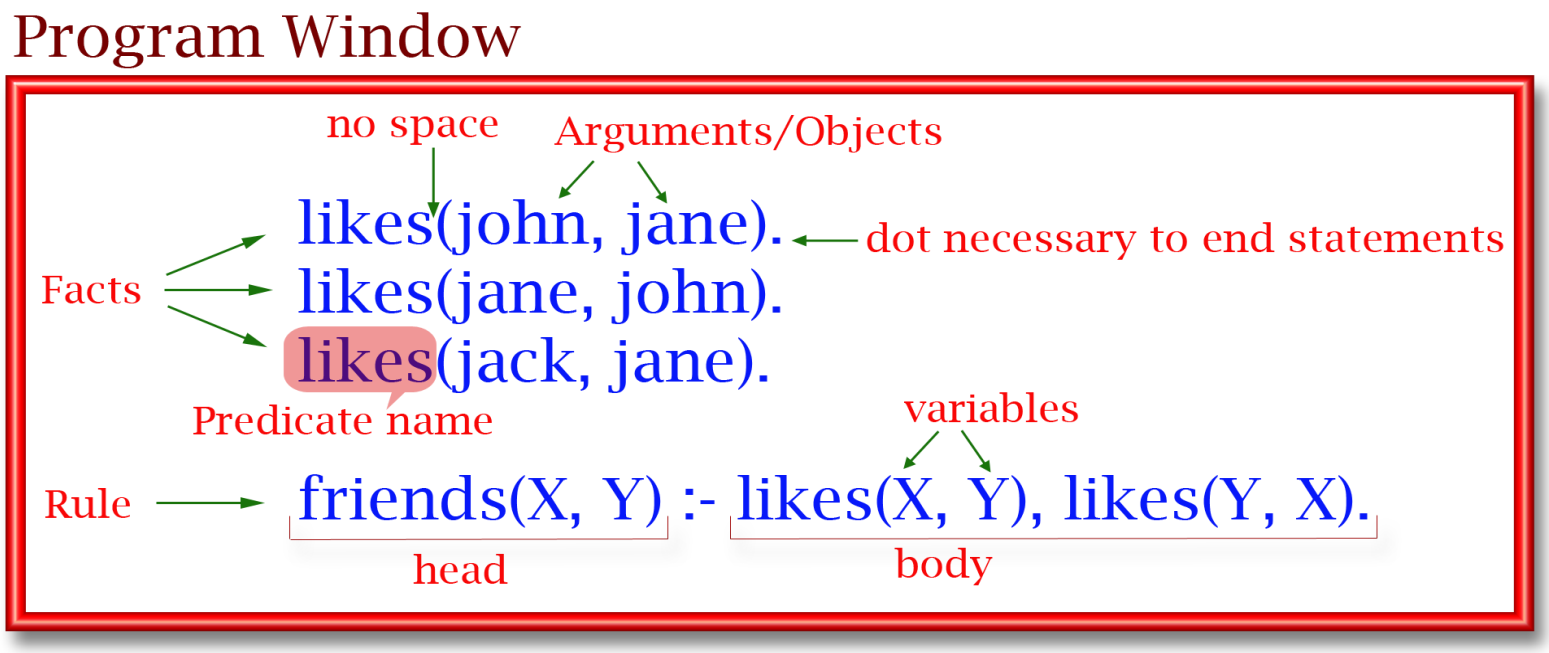


Driven by Functional Call

Programming Paradigm

Declarative Languages

Logic Programming

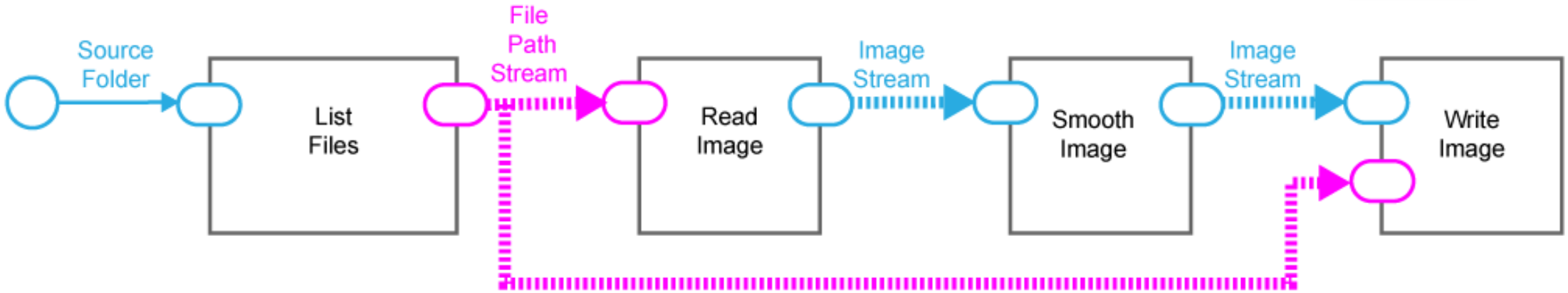


Driven by Logic Reasoning

Programming Paradigm

Declarative Languages

Data Flow Programming



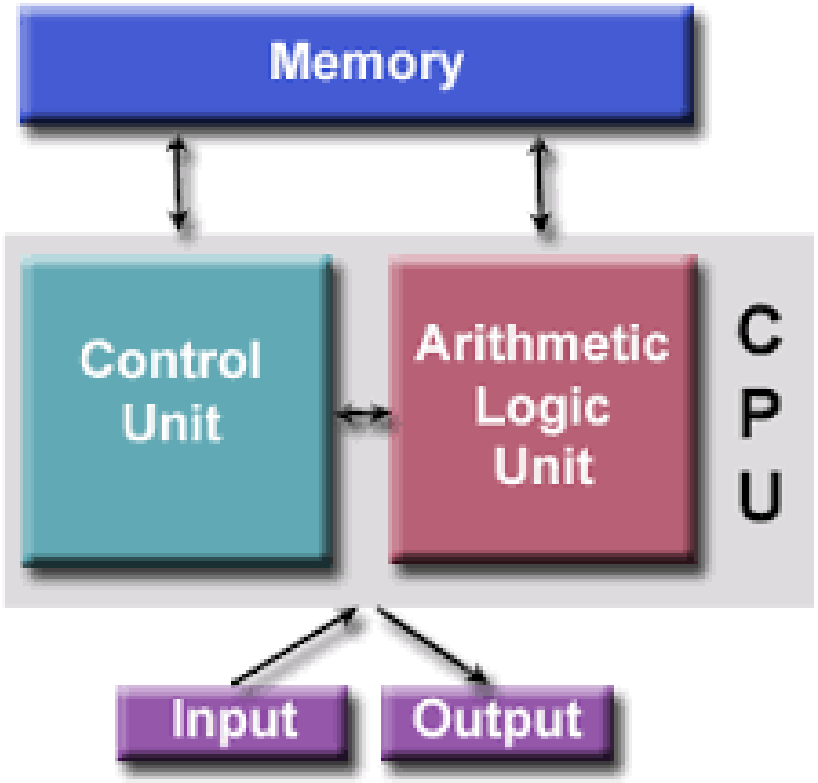
Driven by Data Flow

Programming Paradigm

Imperative Languages

Von Neumann Programming

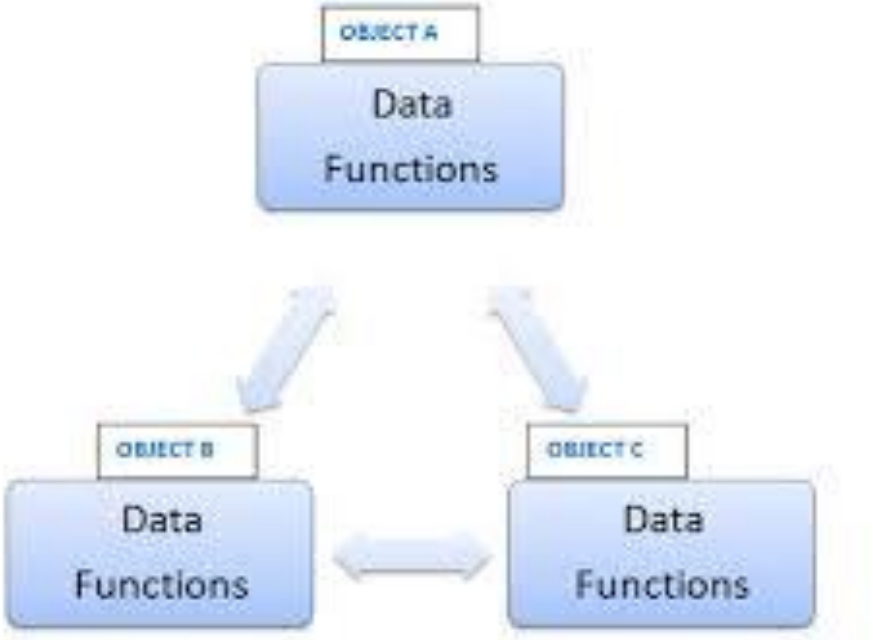
(Accumulator Model)

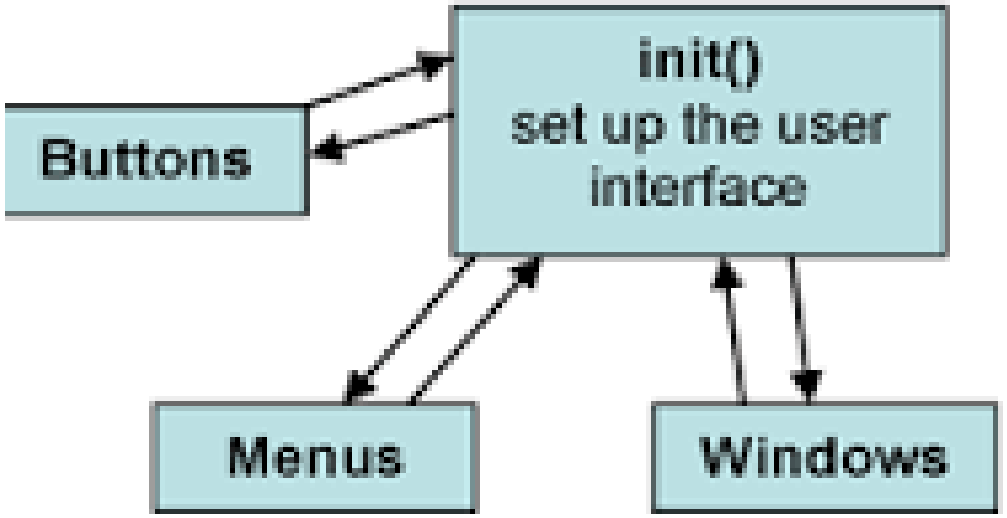


Programming Paradigm

Imperative Languages

Object-Oriented Programming



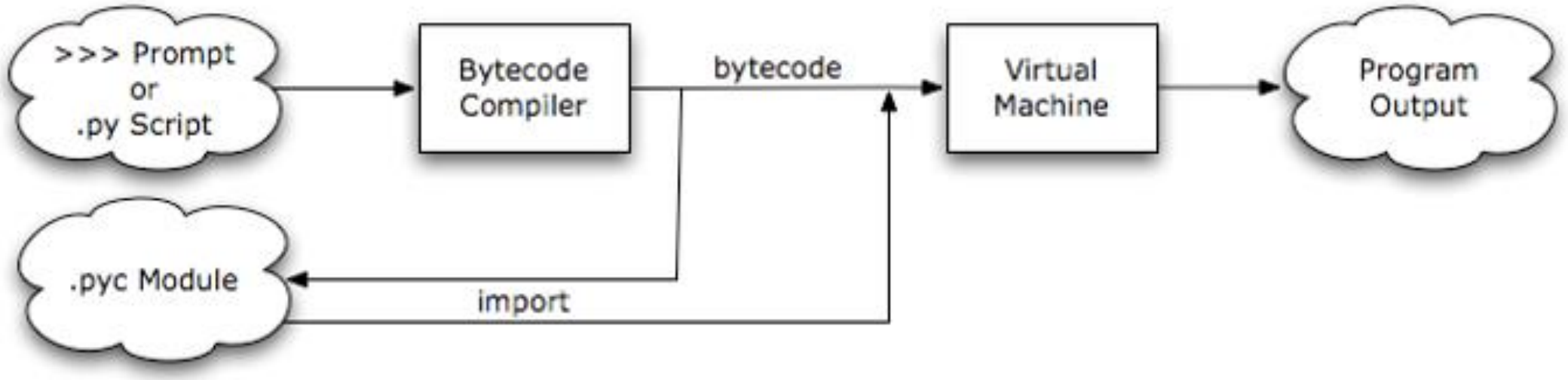


Event-Driven Programming

Programming Paradigm

Imperative Languages

Scripting Programming

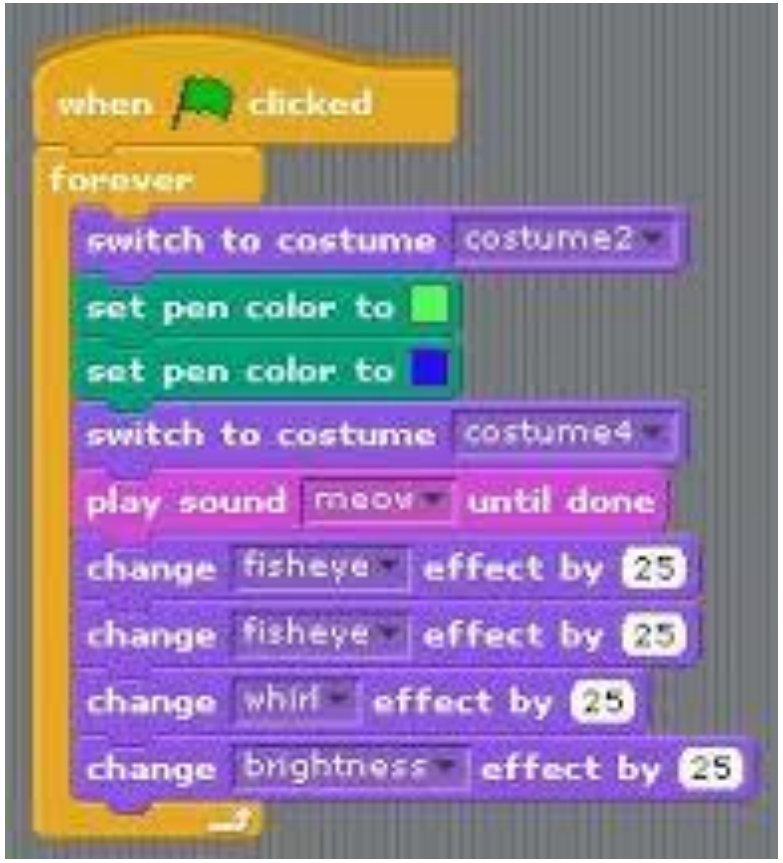


Program on Program

Programming Paradigm

Block Programming

Scratch, Snap!

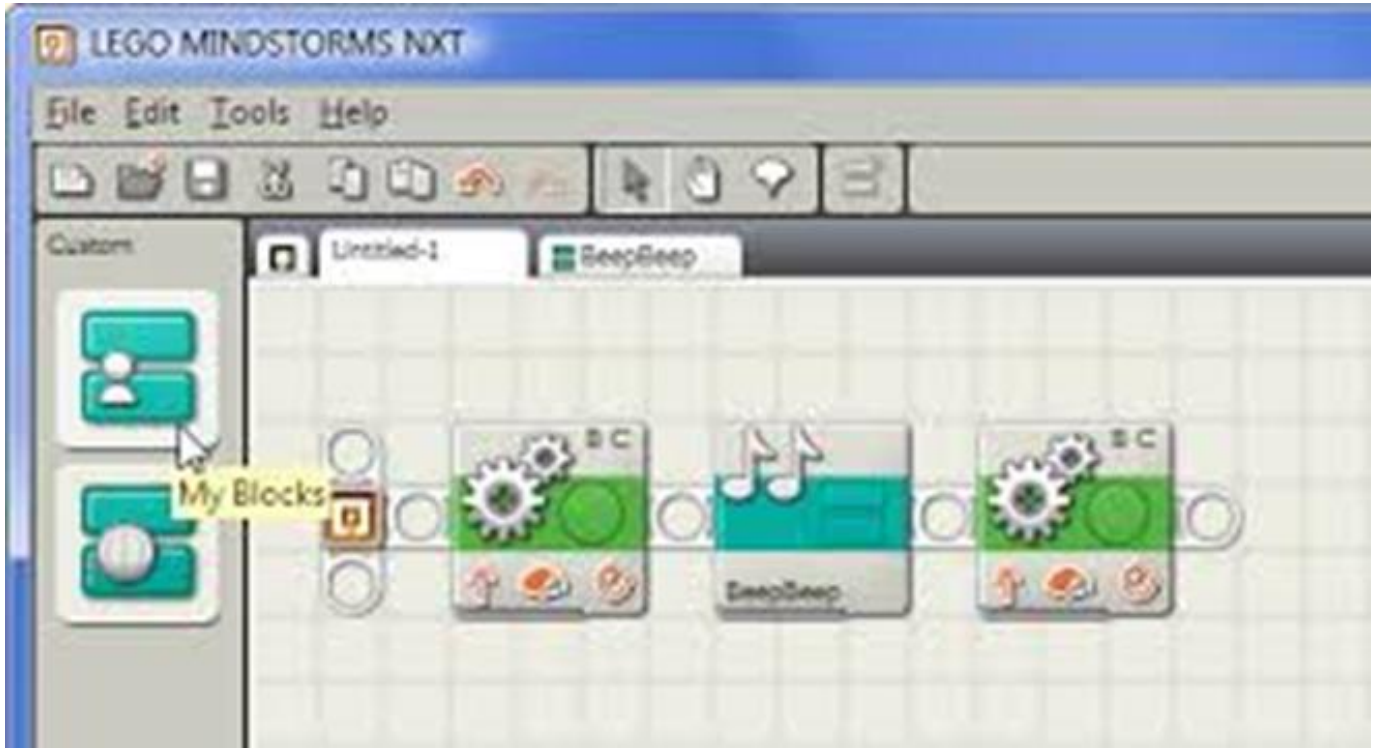


Graphical Design (Imperative, Data Flow, Rule-Based)

Programming Paradigm

Block Programming

My Blocks

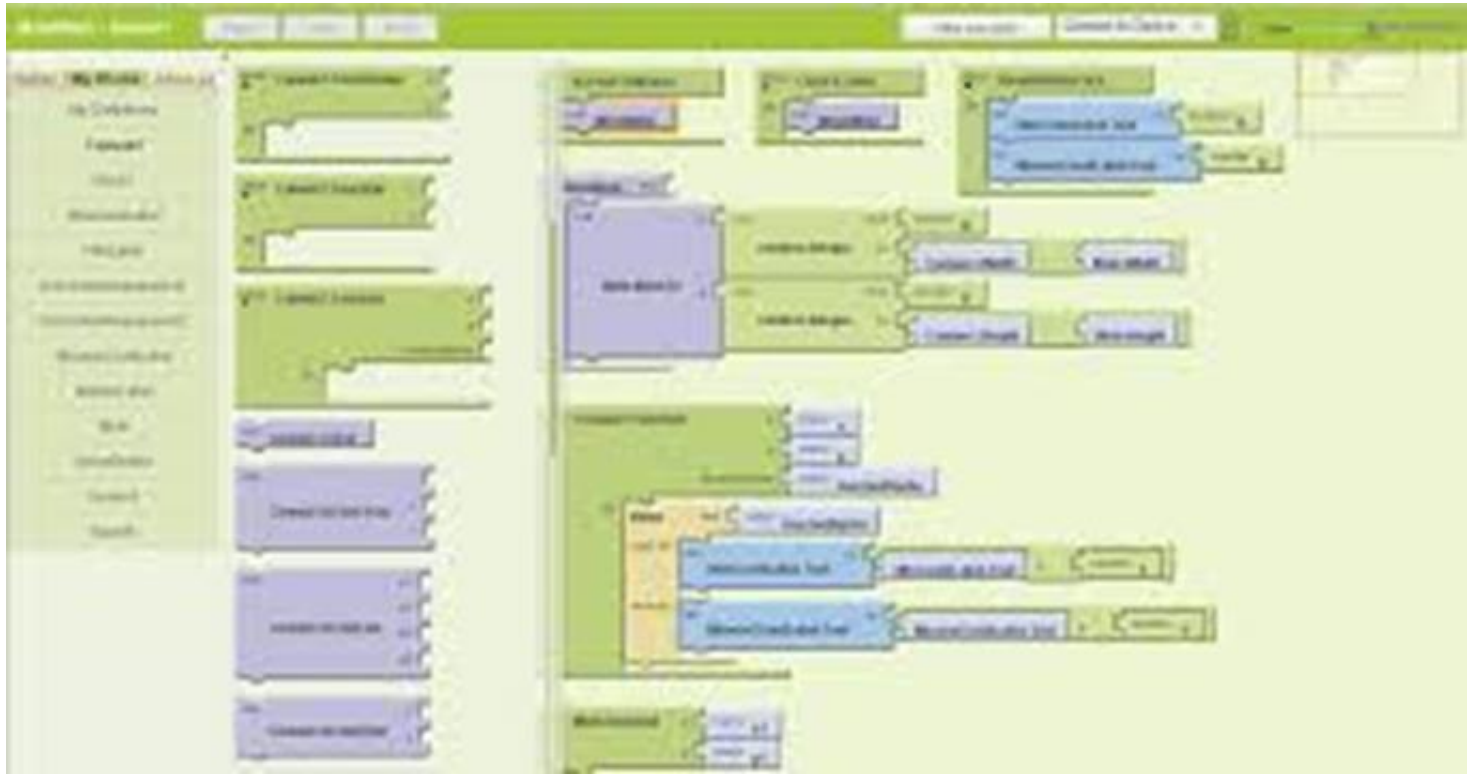


Graphical Design (Imperative, Data Flow, Rule-Based)

Programming Paradigm

Block Programming

MIT My Inventor II



Graphical Design (Imperative, Data Flow, Rule-Based)

```
int gcd(int a, int b) {                                    // C
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}

let rec gcd a b =                                          (* OCaml *)
    if a = b then a
    else if a > b then gcd b (a - b)
         else gcd a (b - a)

gcd(A,B,G) :- A = B, G = A.                                % Prolog
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).
```

Machine

SECTION 4

Von Neumann Machine

John von Neumann

The famous mathematician John von Neumann (1903-1957) was born in a Jewish family in Budapest, Austro-Hungarian Empire, as János Lajos von Neumann. A child-prodigy, János received his Ph.D. in mathematics from Pázmány Péter University in Budapest at the age of 22, simultaneously earning a diploma in chemical engineering from the ETH Zurich in Switzerland. Between 1926 and 1930, he taught as a Privatdozent at the University of Berlin, the youngest in its history. By age 25, he had already published a dozen of major papers.

John von Neumann emigrated to the United States just in time—in 1930, where he was invited to Princeton University, and, subsequently, was one of the first four people selected for the faculty of the Institute for Advanced Study (two of the others being Albert Einstein and Kurt Gödel!), where he remained a mathematics professor from its formation in 1933 until his death.

Von Neumann was an important figure in computer science.

The use of memory in digital computers to store both sequences of instructions and data was a breakthrough to which von Neumann made major contributions.

Von Neumann Machine

In 1945, while consulting for the Moore School of Electrical Engineering on the EDVAC project, von Neumann wrote an incomplete set of notes, titled the First Draft of a Report on the EDVAC. This widely distributed paper laied foundations of a computer architecture in which the data and the program are both stored in the computer's memory in the same address space, which will be described later as von Neumann Architecture (see the lower drawing). This architecture became the de facto standard for a long time and is still used today (until technology enabled more advanced architectures).

Von Neumann also created the field of cellular automata without the aid of computers, constructing the first self-replicating automata with pencil and graph paper. The concept of a universal constructor was fleshed out in his posthumous work Theory of Self Reproducing Automata. Von Neumann proved that the most effective way of performing large-scale mining operations such as mining an entire planet or asteroid belt would be by using self-replicating machines, taking advantage of their exponential growth.

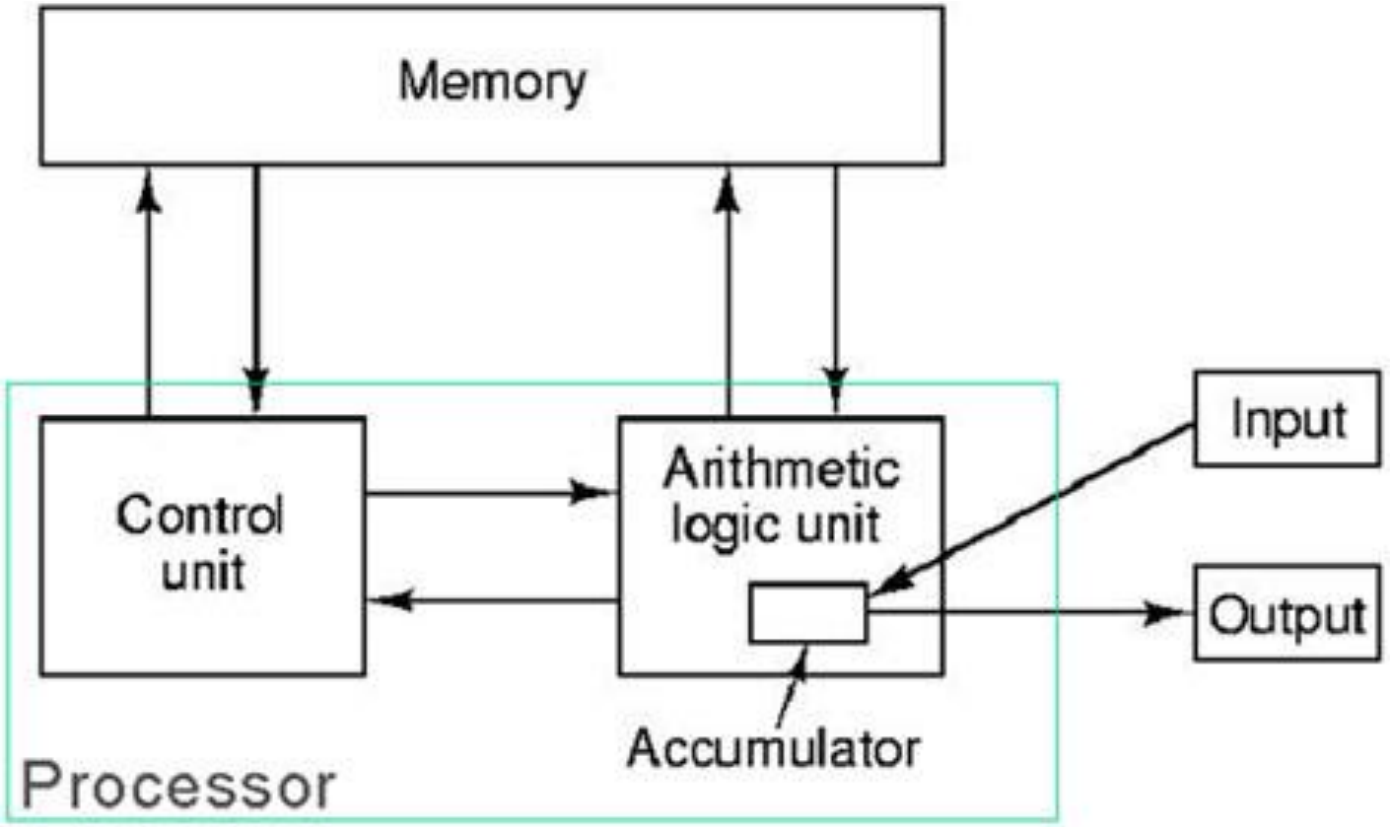
Von Neumann is credited with at least one contribution to the study of algorithms. The renowned computer scientist Donald Knuth cites von Neumann as the inventor (in 1945), of the merge sort algorithm, in which the first and second halves of an array are each sorted recursively and then merged together. His algorithm for simulating a fair coin with a biased coin is used in the software whitening stage of some hardware random number generators.

In 1956 von Neumann wrote his (posthumously published) book The Computer and the Brain, in which discusses how the brain can be viewed as a computing machine. The book is speculative in nature, but discusses several important differences between brains and computers of his day (such as processing speed and parallelism), as well as suggesting directions for future research. **Memory is one of the central themes in his book.**

Imperative Language

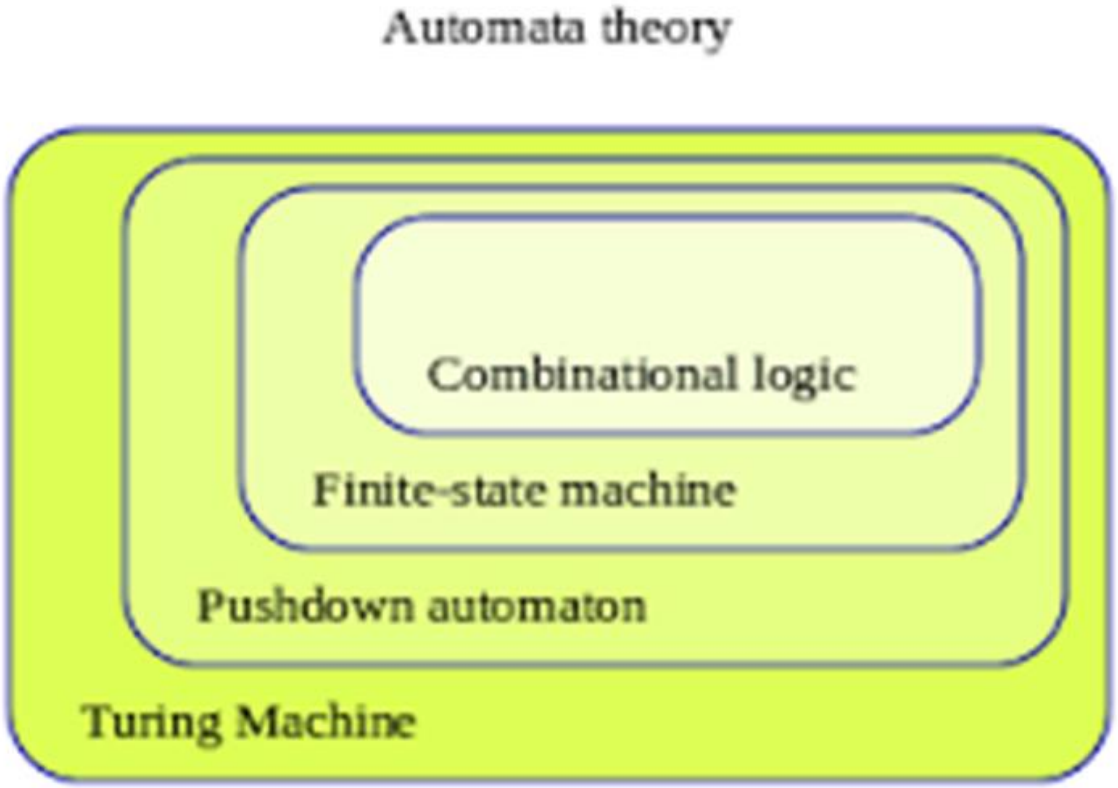
1. Imperative languages, particularly the von Neumann languages, predominate
   - They will occupy the bulk of our attention
2. We also plan to spend a lot of time on functional, logic languages

Von Neumann Machine

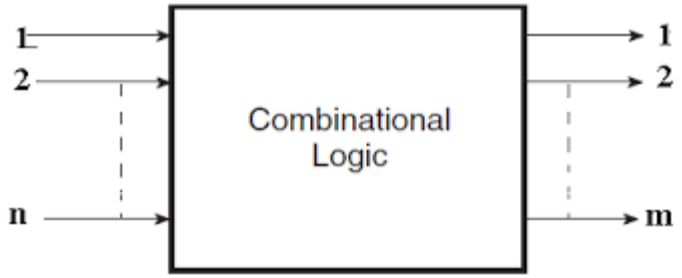


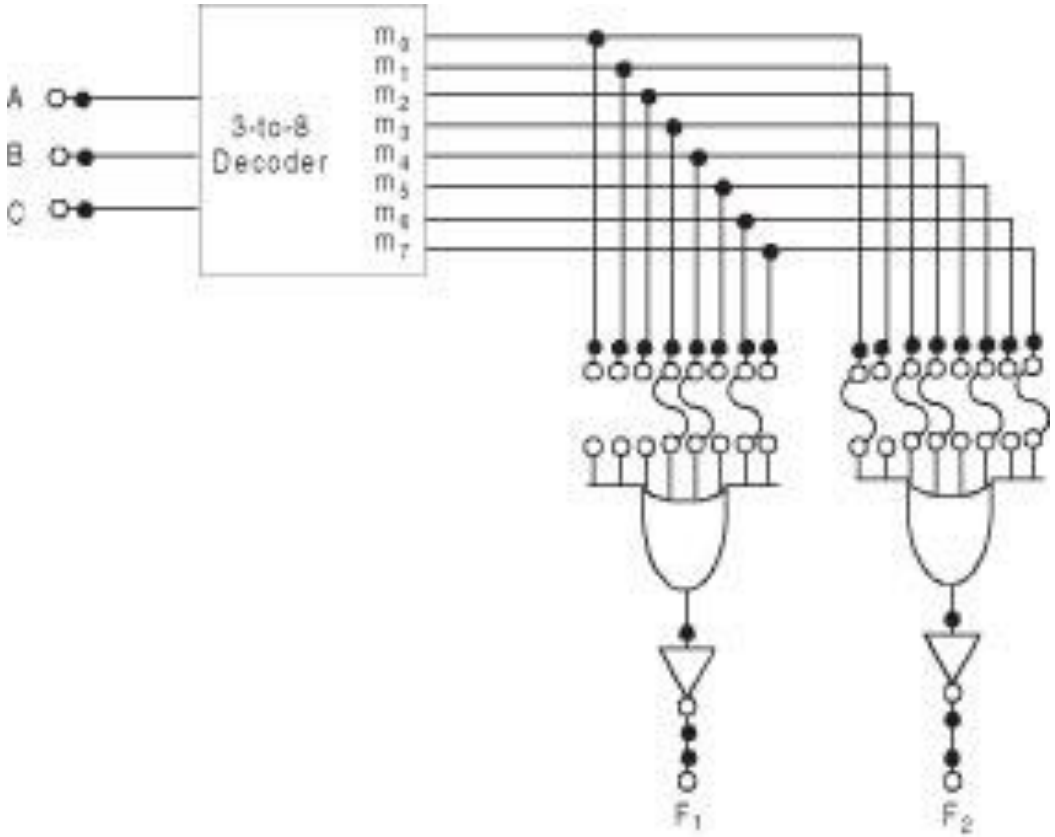
The Computer Architecture for Imperative Languages

Automata Theory

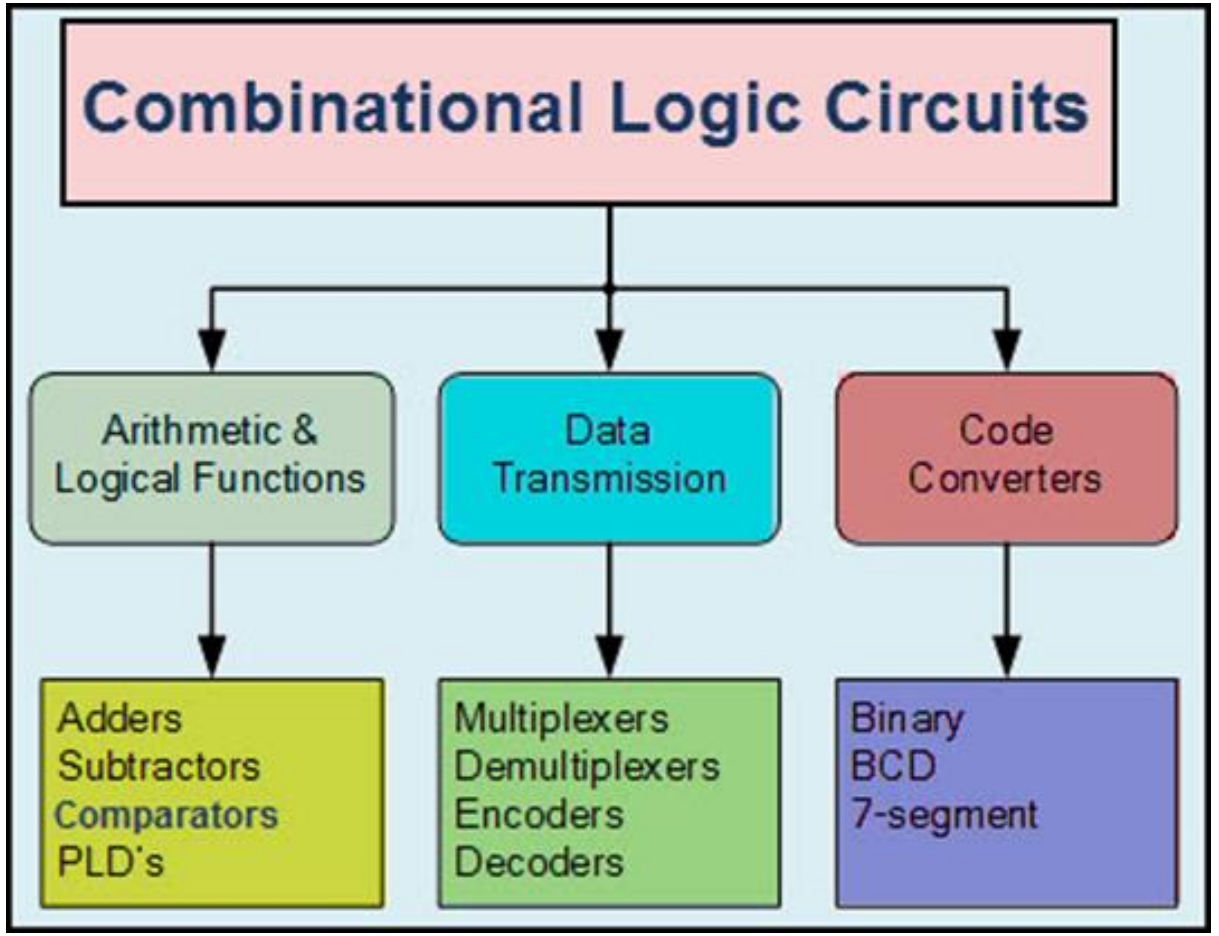


Combinational Logic



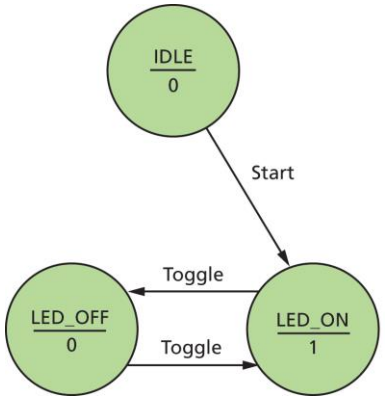


Application for Combinational Logic

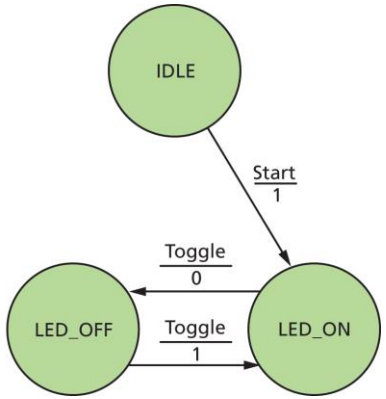


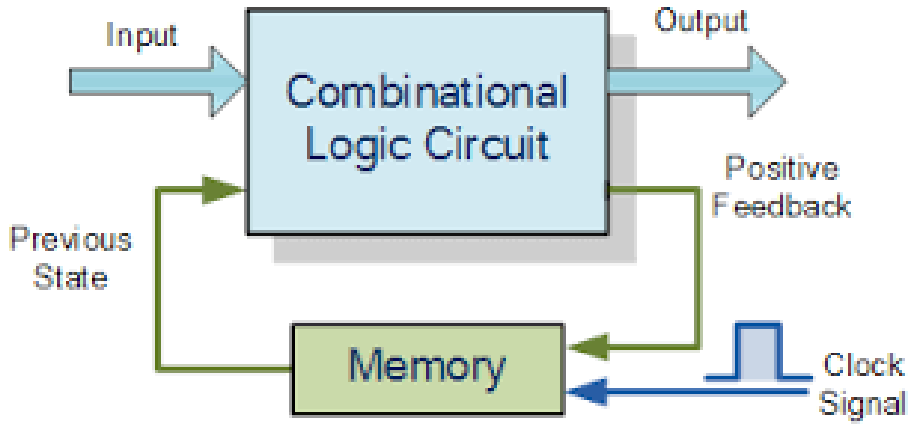
Finite State Machine (FSM)

Synchronous



Asynchronous

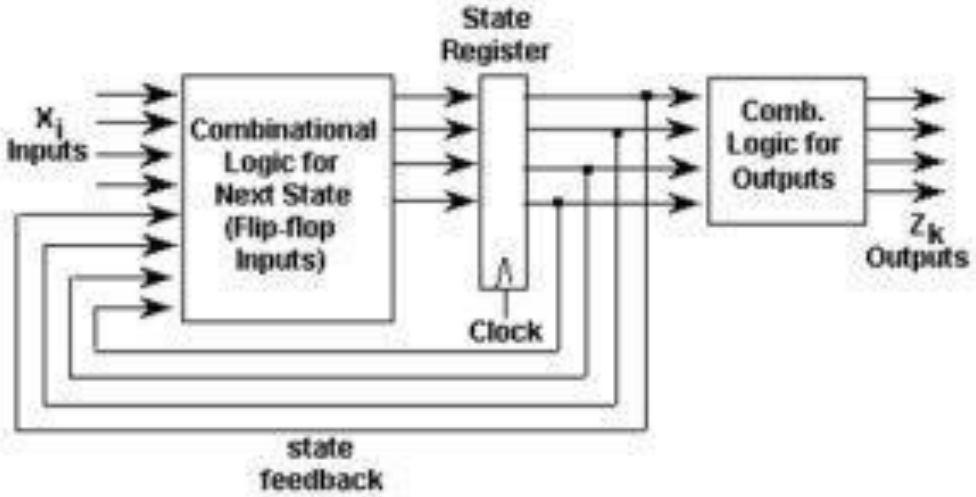




Moore (left) Mealy (right)

Moore and Mealy Machine Design Procedure

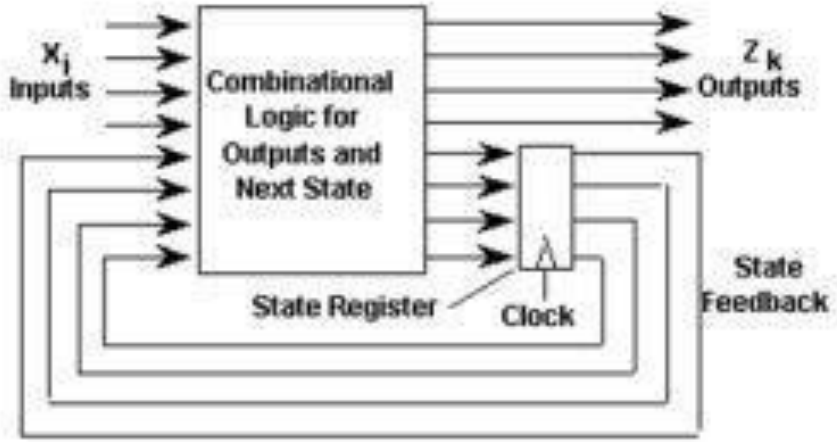
Definitions



Moore Machine

Outputs are function solely of the current state

Outputs change synchronously with state changes



Mealy Machine

Outputs depend on state AND inputs

Input change causes an immediate output change

Asynchronous outputs

Computation

Computing Model
Von Neumann
Rule-based
Fuzzy
Neural
DNA

Programming Paradigm and Architecture
Intel Core i7

Programming Language
C/C++

Interpretation Versus Compilation

SECTION 5

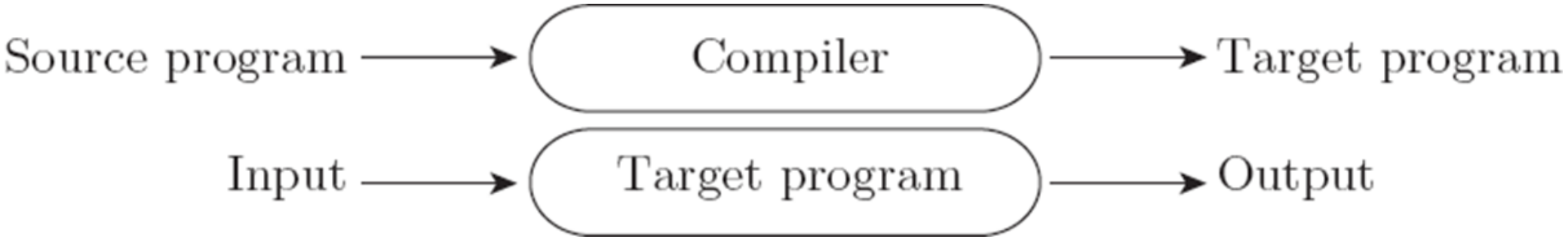
Purpose of Compilation and Interpretation

Convert a language to another language or machine language for execution.

Pure Compilation

C -> Machine Code; C -> Java?

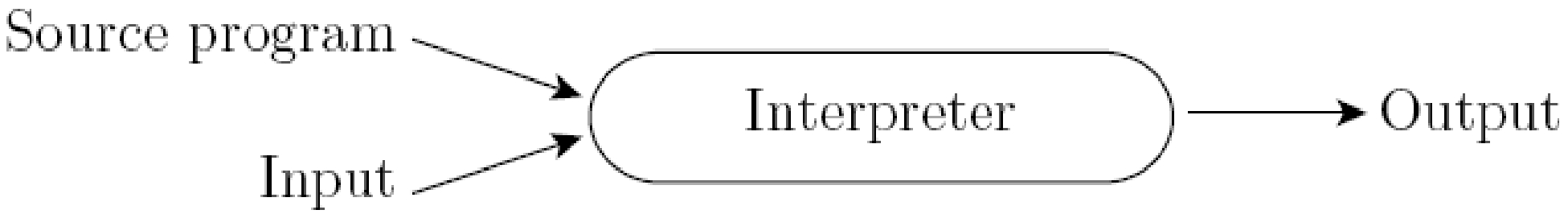
The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



Is it OK to compile C into Java?

Pure Interpretation

1. Interpreter stays around for the execution of the program
2. Interpreter is the locus of control during execution



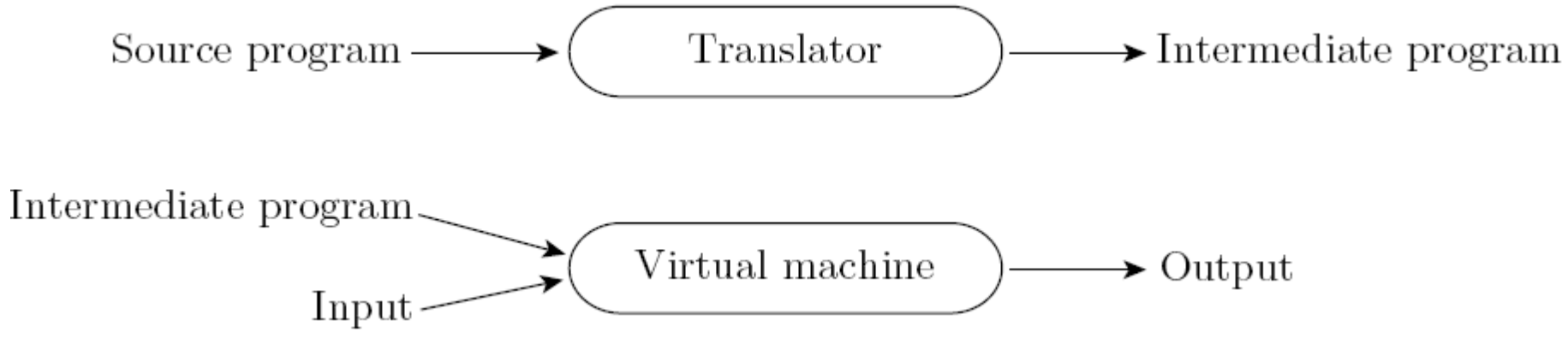
Comparison

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)

- Compilation
  - Better performance

Hybrids

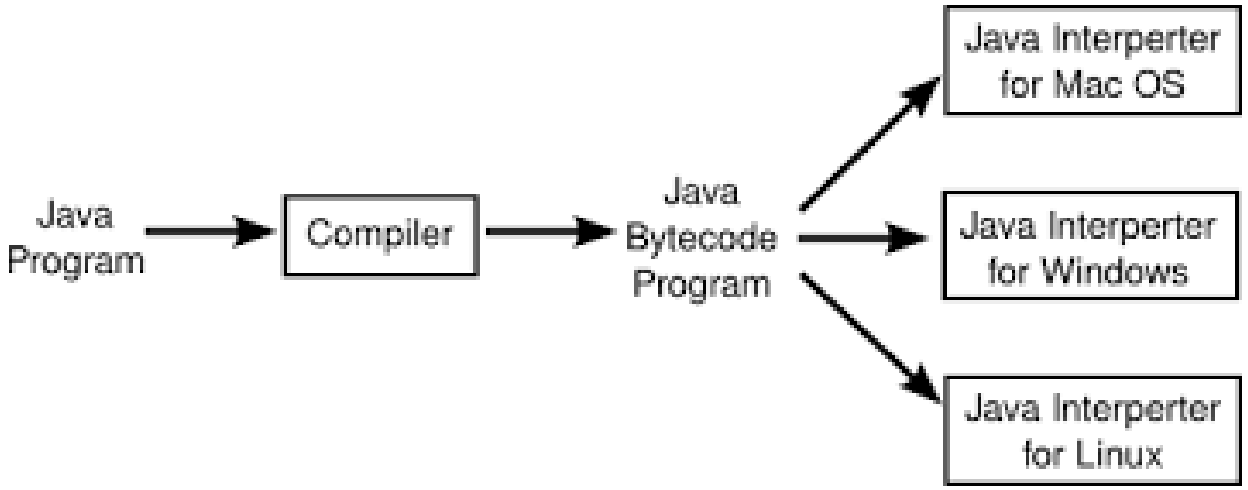
- Common case is compilation or simple pre-processing, followed by interpretation
- Most modern language implementations include a mixture of both compilation and interpretation

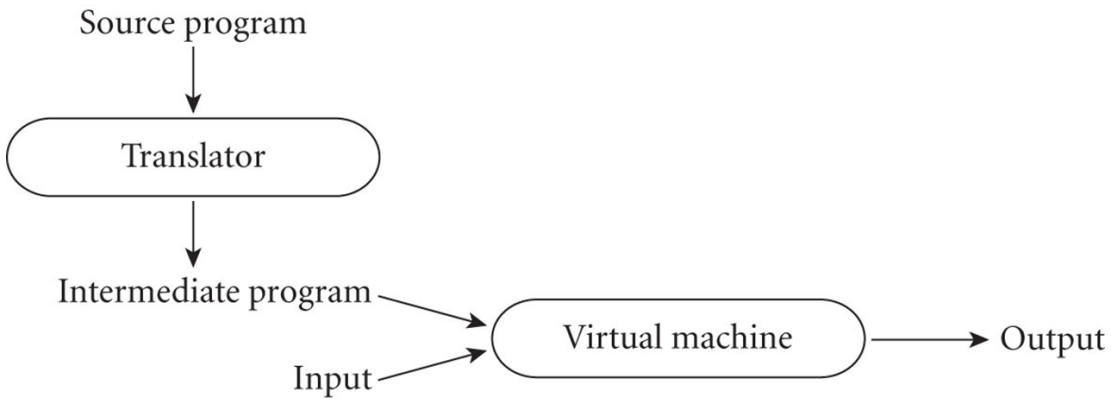


Virtual Machines

Step 1: Compilation from Java to Byte Code

Step 2: Executing the byte code on a machine with native codes





Notes:

- Note that compilation does NOT have to produce machine language for some sort of hardware
- Compilation is translation from one language into another, with full analysis of the meaning of the input
- Compilation entails semantic understanding of what is being processed; pre-processing does not
- A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

Notes:

- Many compiled languages have interpreted pieces, e.g., formats in Fortran or C
- Most use “virtual instructions”
  - set operations in Pascal
  - string manipulation in Basic
- Some compilers produce nothing but virtual instructions, e.g., Pascal P-code, Java byte code, Microsoft COM+

Compilation Strategy
(Mixed Compilation/Interpretation)

SECTION 6

Compilation Strategies

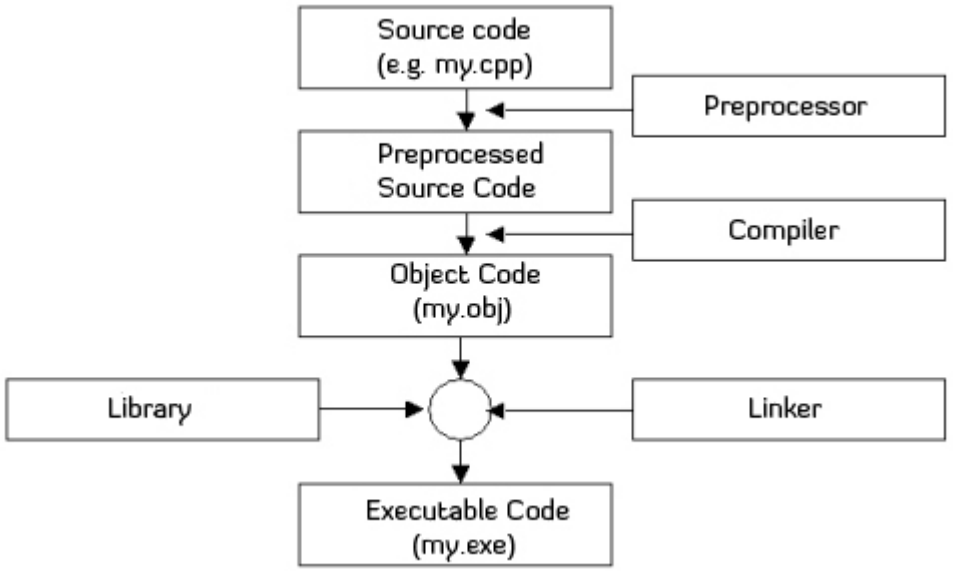
- Every language may use one or more compilation strategies.
- Examples are shown for certain language, but it may not be the only language to use that strategy.

(A) Preprocessor

Tokens

- Removes comments and white space
- Groups characters into tokens (keywords, identifiers, numbers, symbols)
- Expands abbreviations in the style of a macro assembler
- Identifies higher-level syntactic structures (loops, subroutines)

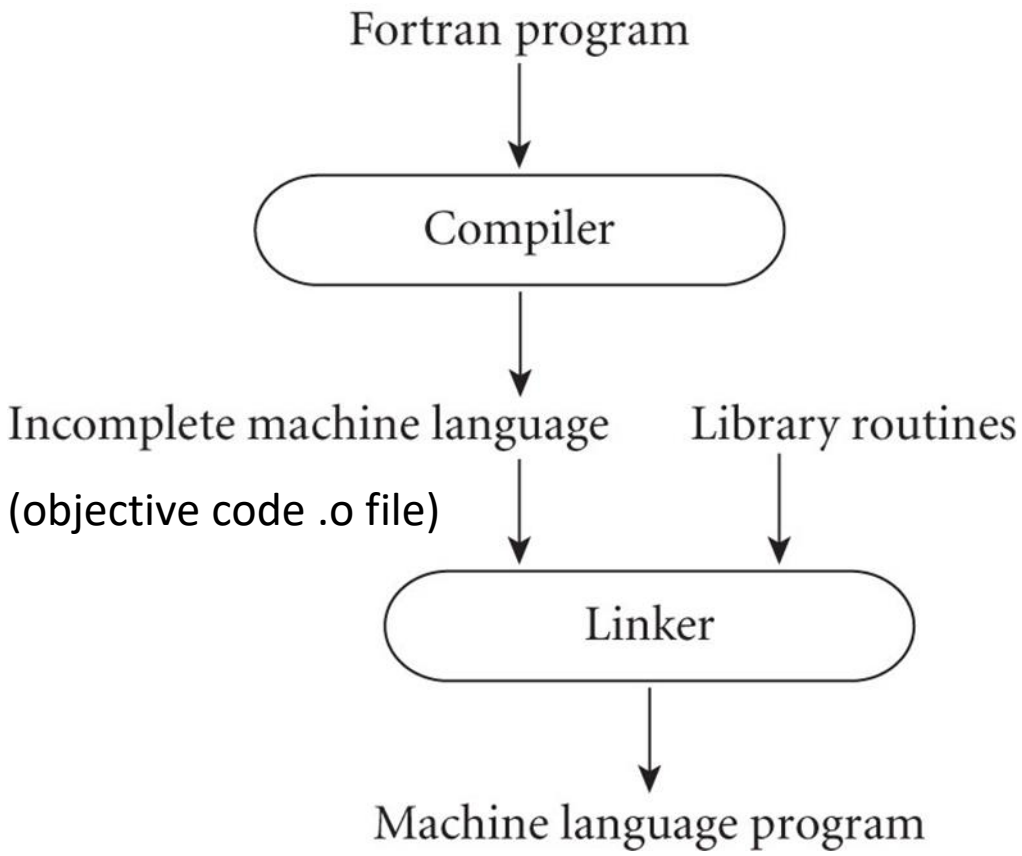
C/C++ Preprocessor



Different preprocessor directives (commands) perform different tasks. We can categorize the Preprocessor Directives as follows:

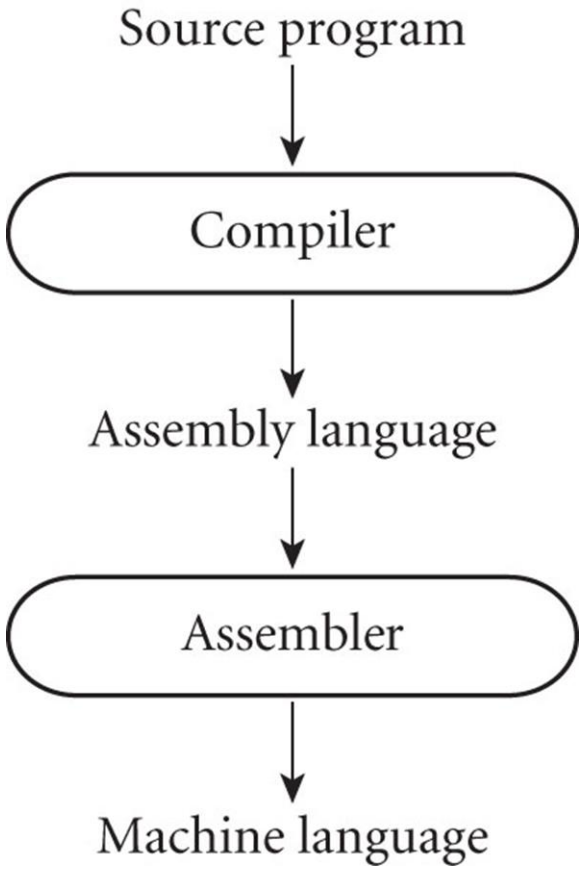
- Inclusion Directives **(#include)**
- Macro Definition Directives **(#define #undef)**
- Conditional Compilation Directives **(#if, #elif, #endif, #ifdef, #ifndef)**
- Other Directives **(#error, #line, #pragma)**

(B) Library of Routines and Linking



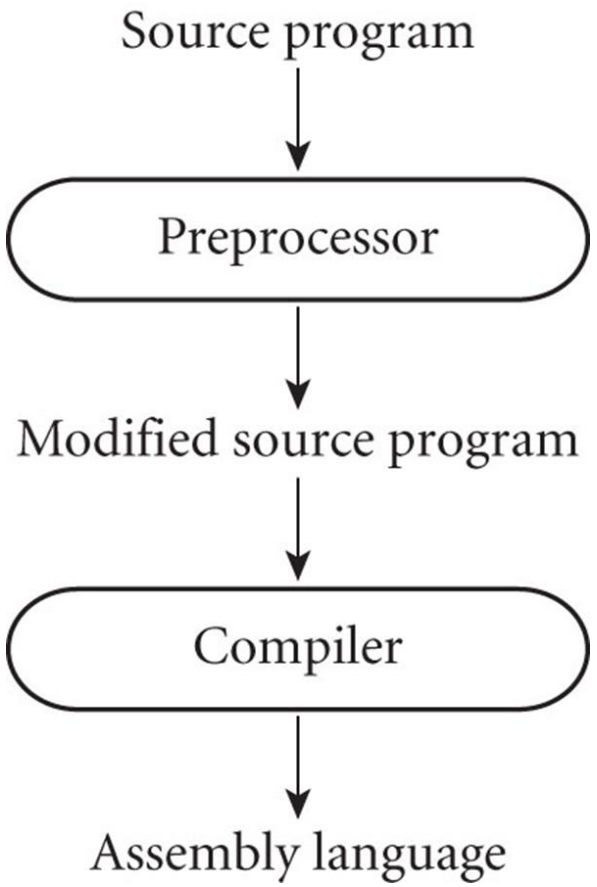
- Compiler uses a linker program to merge the appropriate library of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:

(C)Post-compilation Assembly



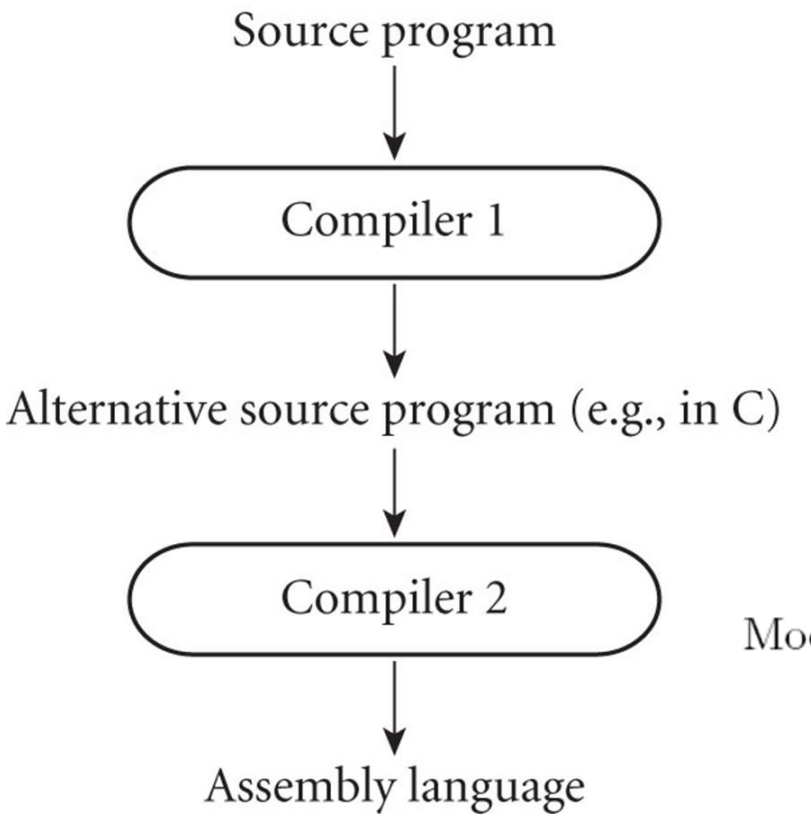
- Facilitates debugging (assembly language easier for people to read)
- Isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers)

(D) The C Preprocessor (conditional compilation)

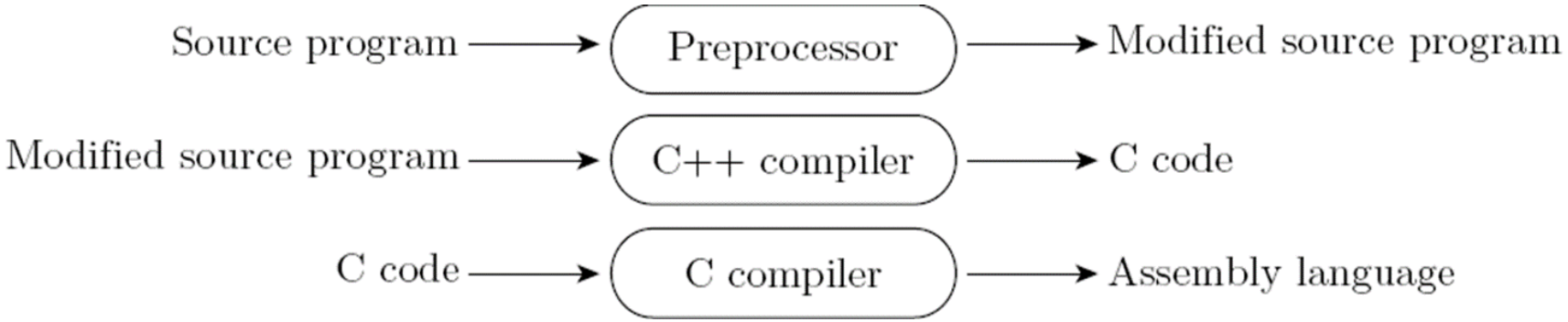


- Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source

(E) Source-to-Source Translation (C++)



- C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language: (combination of A and E)

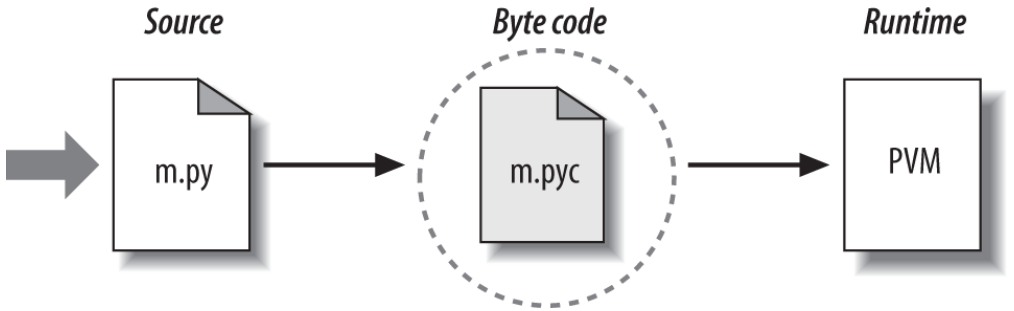


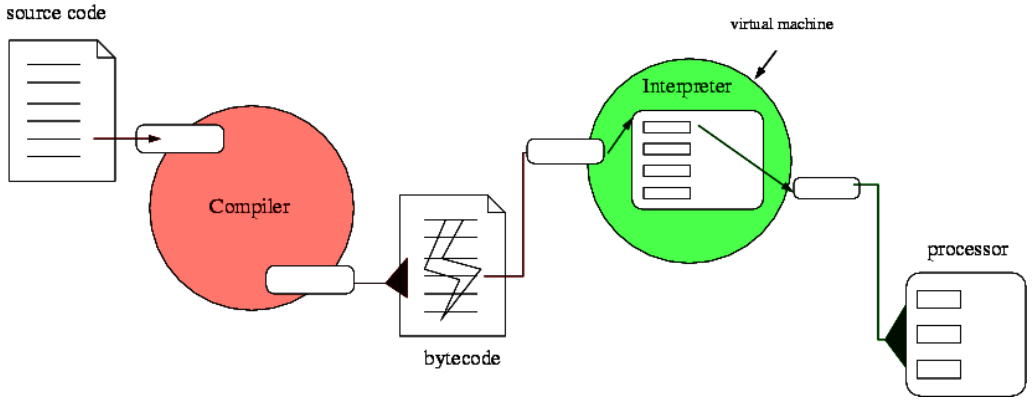
Interpretation Strategy

SECTION 7

(F) Compilation of Interpreted Languages

- The compiler generates code that makes assumptions about decisions that won't be finalized until runtime. If these assumptions are valid, the code runs very fast. If not, a dynamic check will revert to the interpreter.

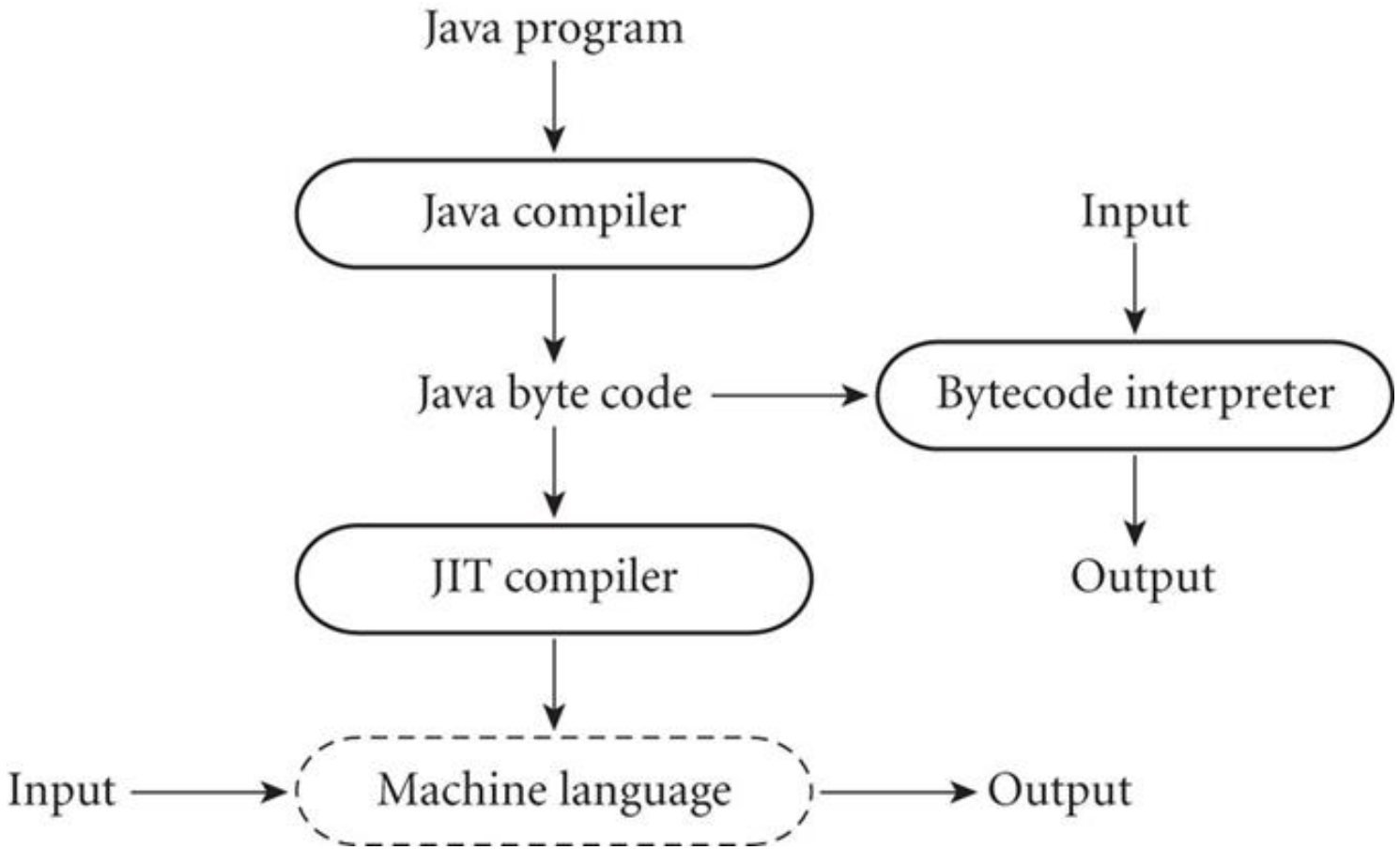


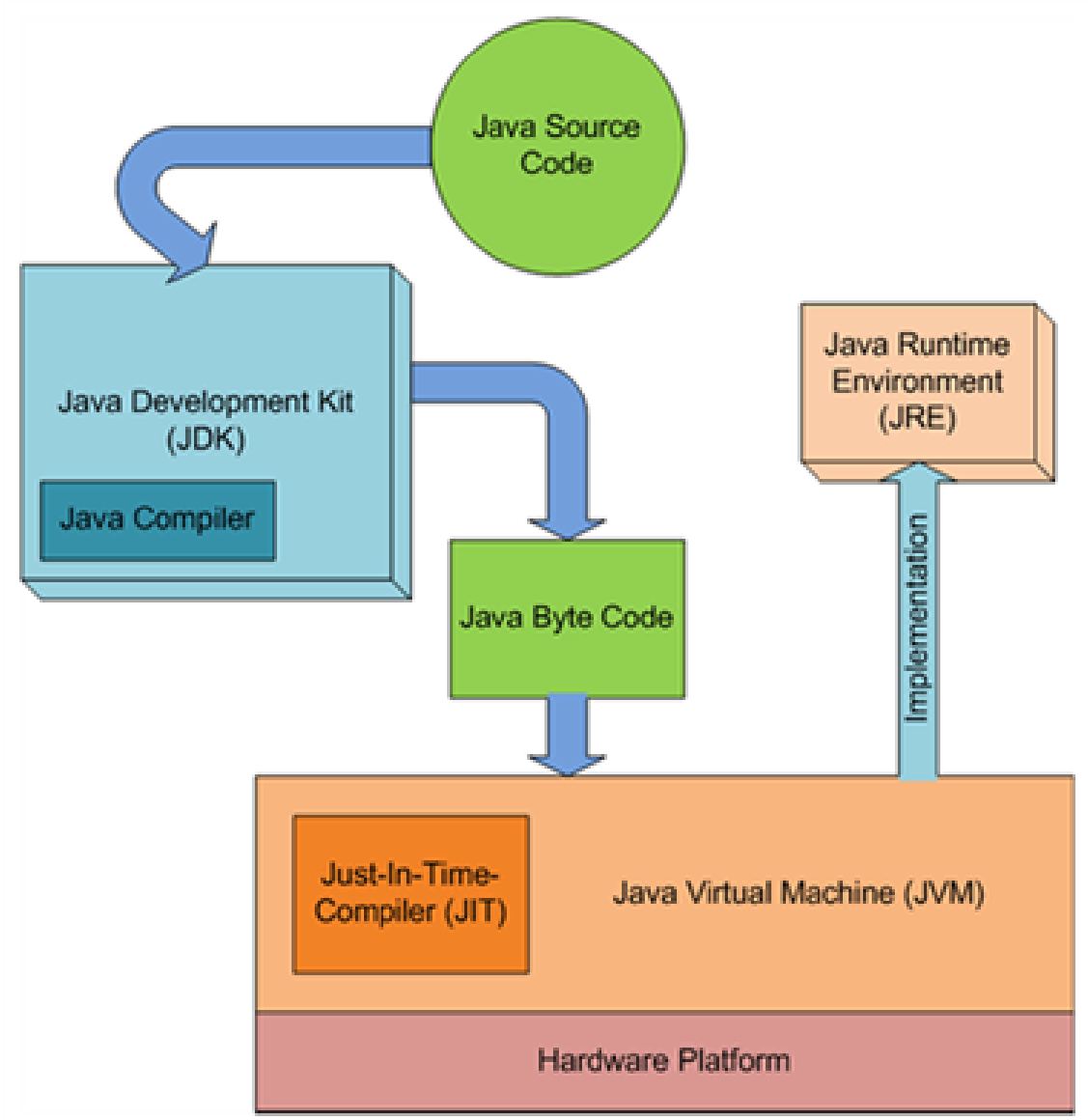


(G) Dynamic and Just-in-Time Compilation

- In some cases a programming system may deliberately delay compilation until the last possible moment.
- Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.
- The Java language definition defines a machine-independent intermediate form known as byte code. Byte code is the standard format for distribution of Java programs.
- The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

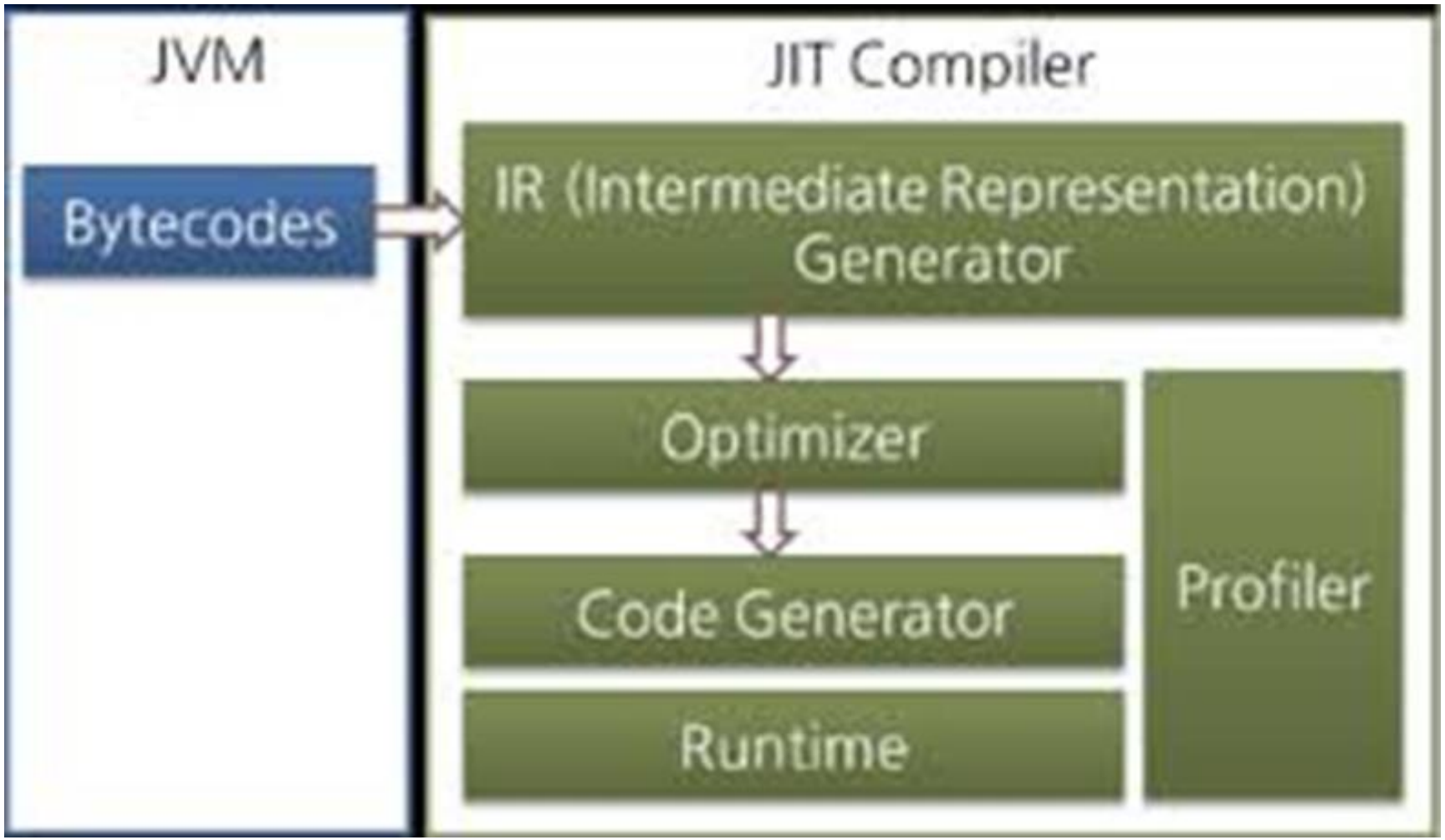
Java JIT

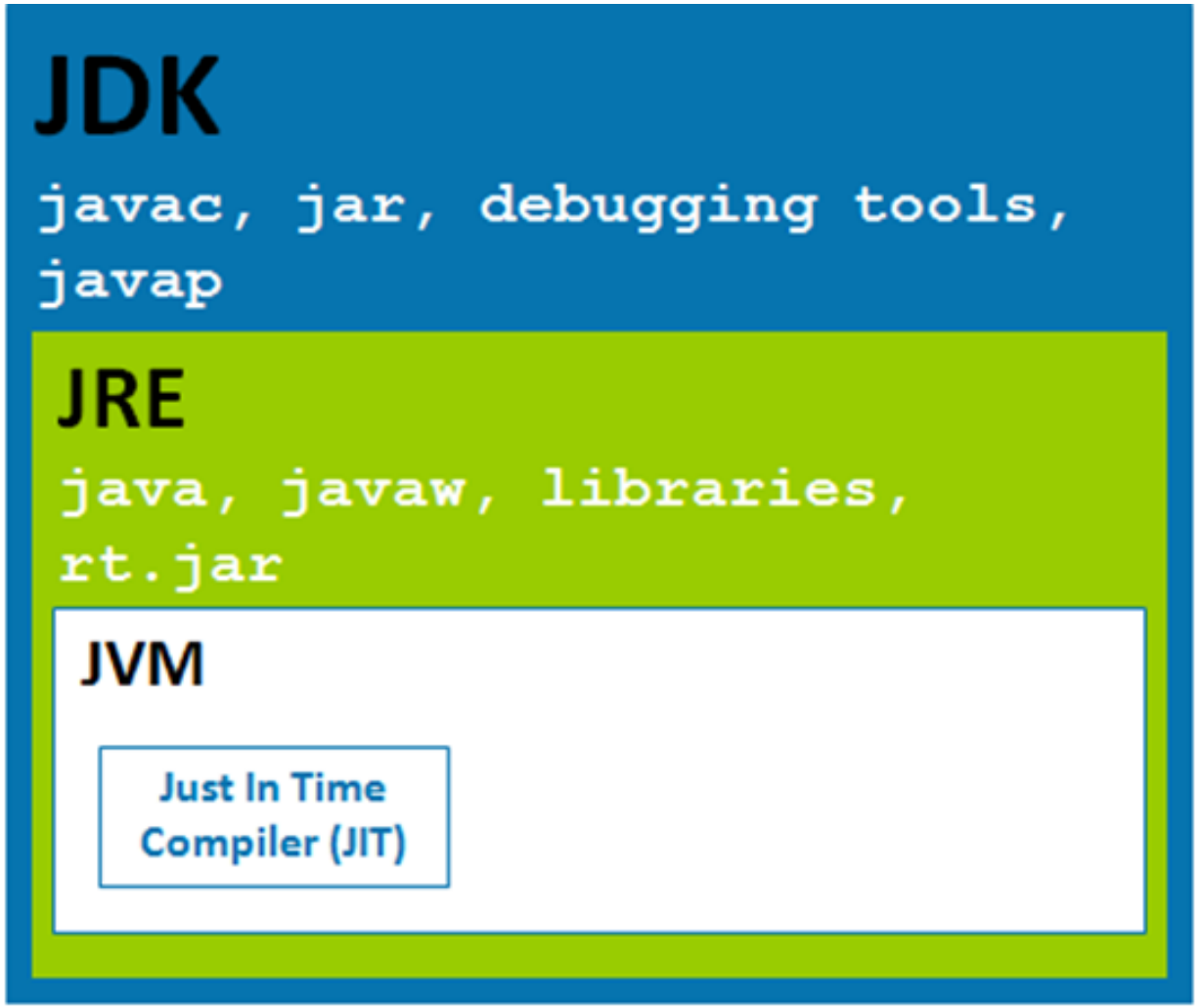

Java program
Java compiler
Java byte code
JIT compiler
Machine language
Input
Output
Input
Bytecode interpreter
Output



Just-In Time Compilation

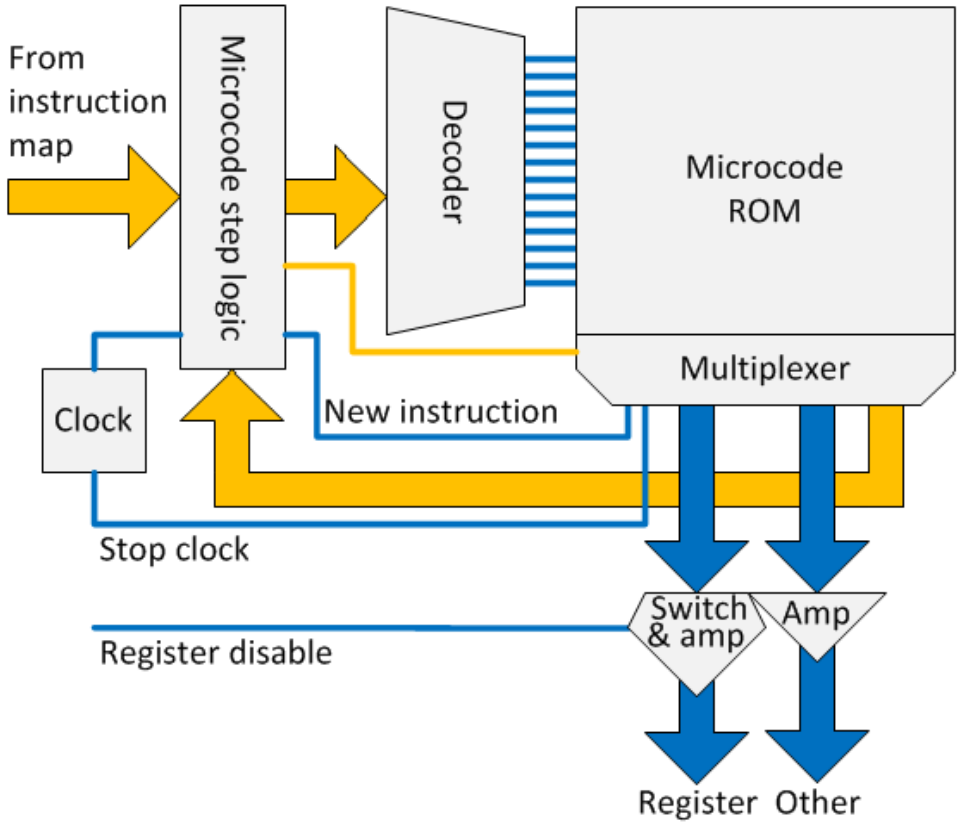
Wait until you can no longer wait!


JVM
Bytecodes
JIT Compiler
IR (Intermediate Representation) Generator
Optimizer
Code Generator
Runtime
Profiler


JDK
javac, jar, debugging tools, javap
JRE
java, javaw, libraries, rt.jar
JVM
Just In Time Compiler (JIT)

(H) Microcode

Instruction is Translated to Microcode

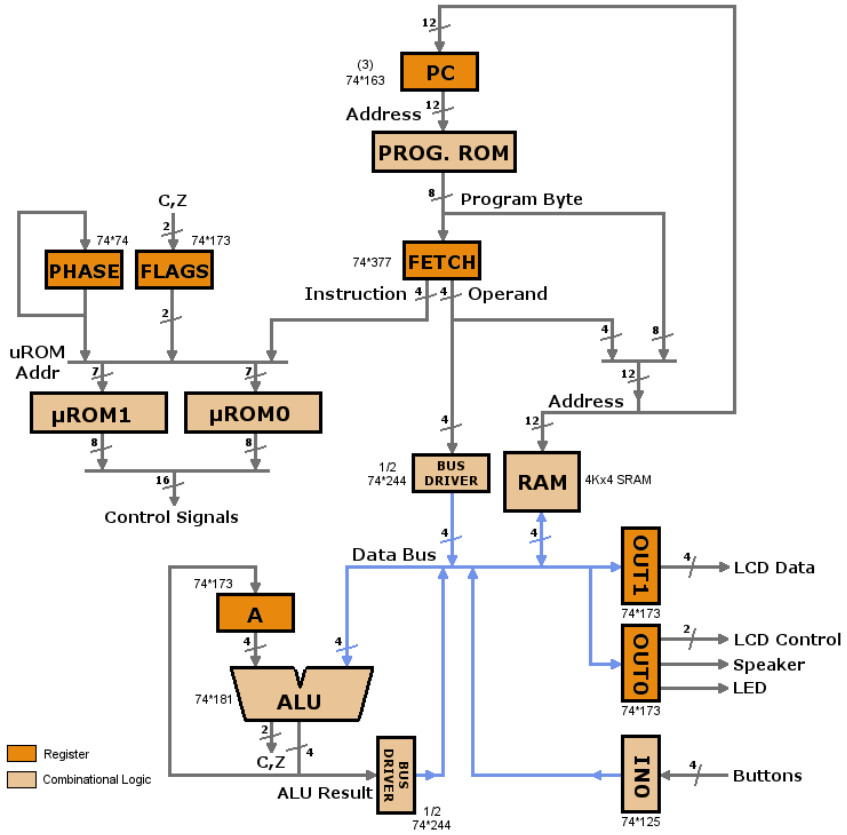


- Assembly-level instruction set is not implemented in hardware; it runs on an interpreter.
- Interpreter is written in low-level instructions (microcode or firmware), which are stored in read-only memory and executed by the hardware.

Nibbler 4 Bit CPU

Example for Microcode

Datapath



Instruction to Microcode

instruction	i3	i2	i1	i0	/C	/Z	phase	incPC	/loadPC	/loadA	/loadFlags	/carryIn	M	S3	S2	S1	S0	/csRAM	/weRAM	/oeALU	/oeIN	/oeOprnd	/loadOut
any	x	x	x	x	x	x	0	1	1	1	1	1	0	0	0	0	0	1	1	0	1	1	1
JC	0	0	0	0	0	x	1	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1
JC	0	0	0	0	1	x	1	1	1	1	1	1	0	0	0	0	0	1	1	0	1	1	1
JNC	0	0	0	1	0	x	1	1	1	1	1	1	0	0	0	0	0	1	1	0	1	1	1
JNC	0	0	0	1	1	x	1	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1
CMPI	0	0	1	0	x	x	1	0	1	1	0	0	0	0	1	1	0	1	1	1	1	0	1
CMPM	0	0	1	1	x	x	1	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1
LIT	0	1	0	0	x	x	1	0	1	0	0	dc	1	1	0	1	0	1	1	1	1	0	1
IN	0	1	0	1	x	x	1	0	1	0	0	dc	1	1	0	1	0	1	1	1	0	1	1
LD	0	1	1	0	x	x	1	1	1	0	0	dc	1	1	0	1	0	0	1	1	1	1	1
ST	0	1	1	1	x	x	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1
JZ	1	0	0	0	x	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1
JZ	1	0	0	0	x	1	1	1	1	1	1	1	0	0	0	0	0	1	1	0	1	1	1
JNZ	1	0	0	1	x	0	1	1	1	1	1	1	0	0	0	0	0	1	1	0	1	1	1
JNZ	1	0	0	1	x	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1
ADDI	1	0	1	0	x	x	1	0	1	0	0	1	0	1	0	0	1	1	1	1	1	0	1
ADDM	1	0	1	1	x	x	1	1	1	0	0	1	0	1	0	0	1	0	1	1	1	1	1
JMP	1	1	0	0	x	x	1	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1
OUT	1	1	0	1	x	x	1	0	1	1	1	1	0	0	0	0	0	1	1	0	1	1	0
NORI	1	1	1	0	x	x	1	0	1	0	0	dc	1	0	0	0	1	1	1	1	1	0	1
NORM	1	1	1	1	x	x	1	1	1	0	0	dc	1	0	0	0	1	0	1	1	1	1	1

Notes:

- Compilers exist for some interpreted languages, but they aren't pure:
  - selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source.
  - Interpretation of parts of code, at least, is still necessary for reasons above.
- Unconventional compilers
  - text formatters
  - silicon compilers
  - query language processors

Programming Environment

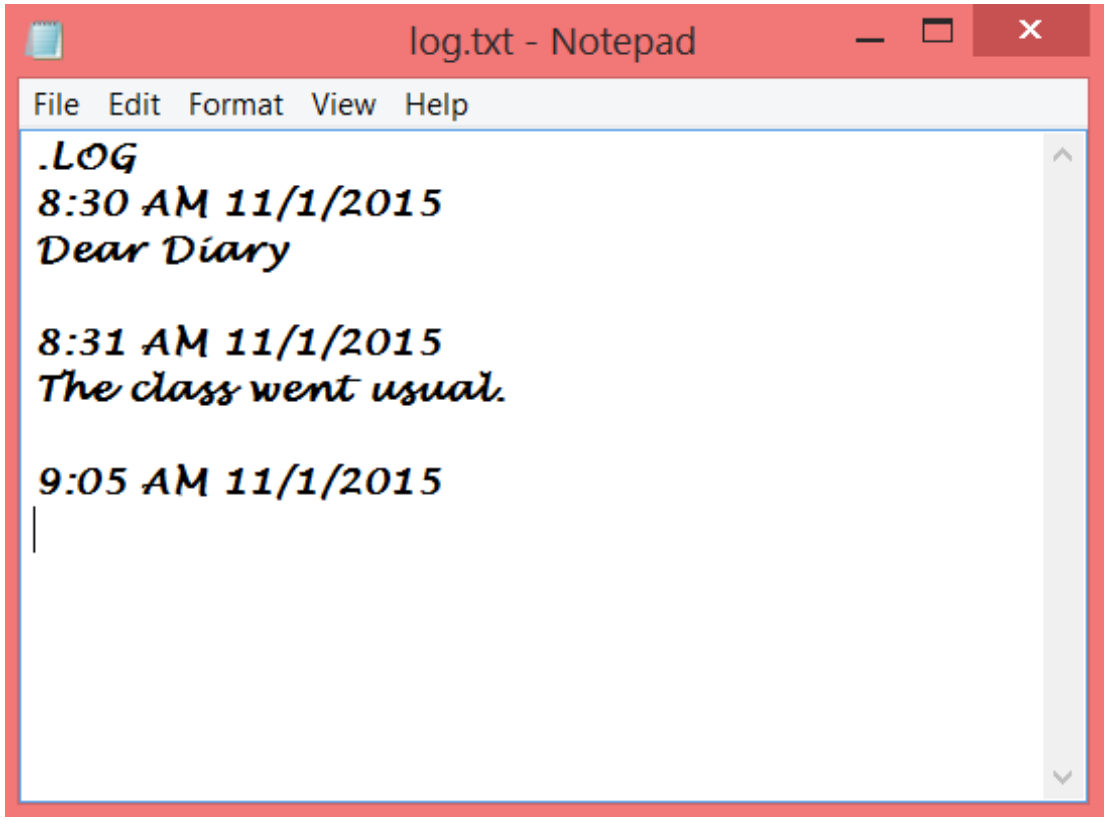
SECTION 8

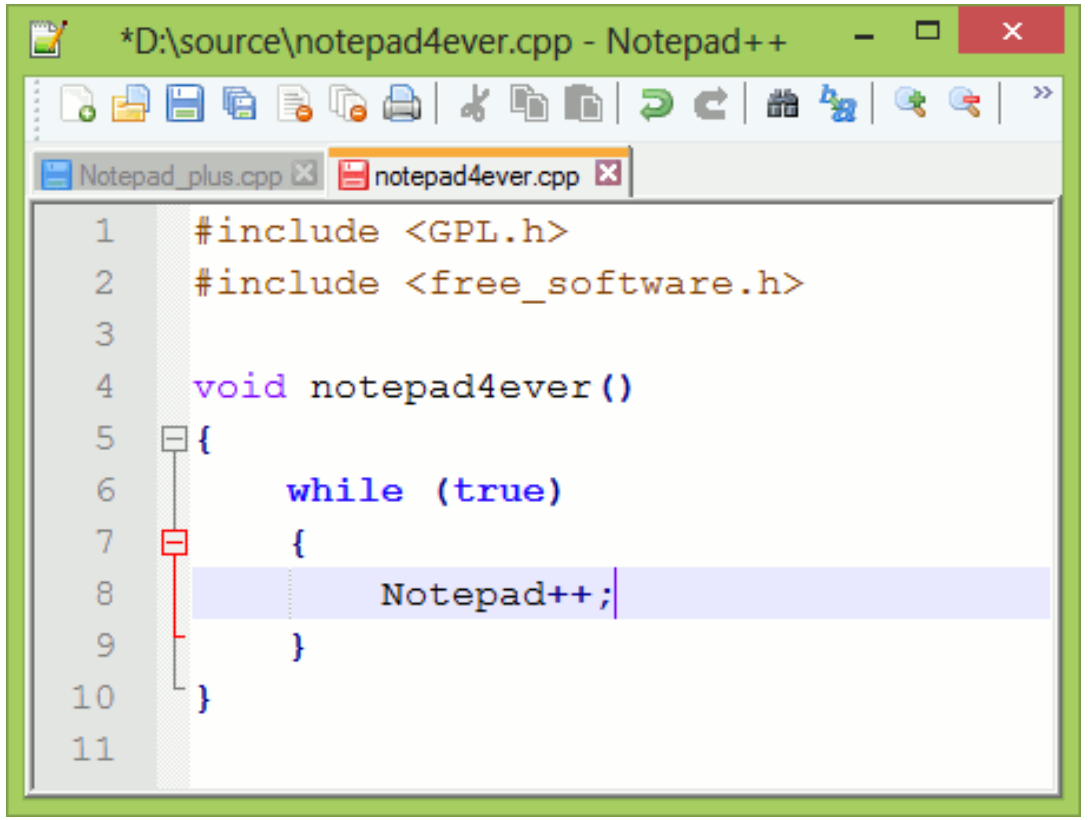
Unix Tools

Still Widely Used in Linux-based Systems

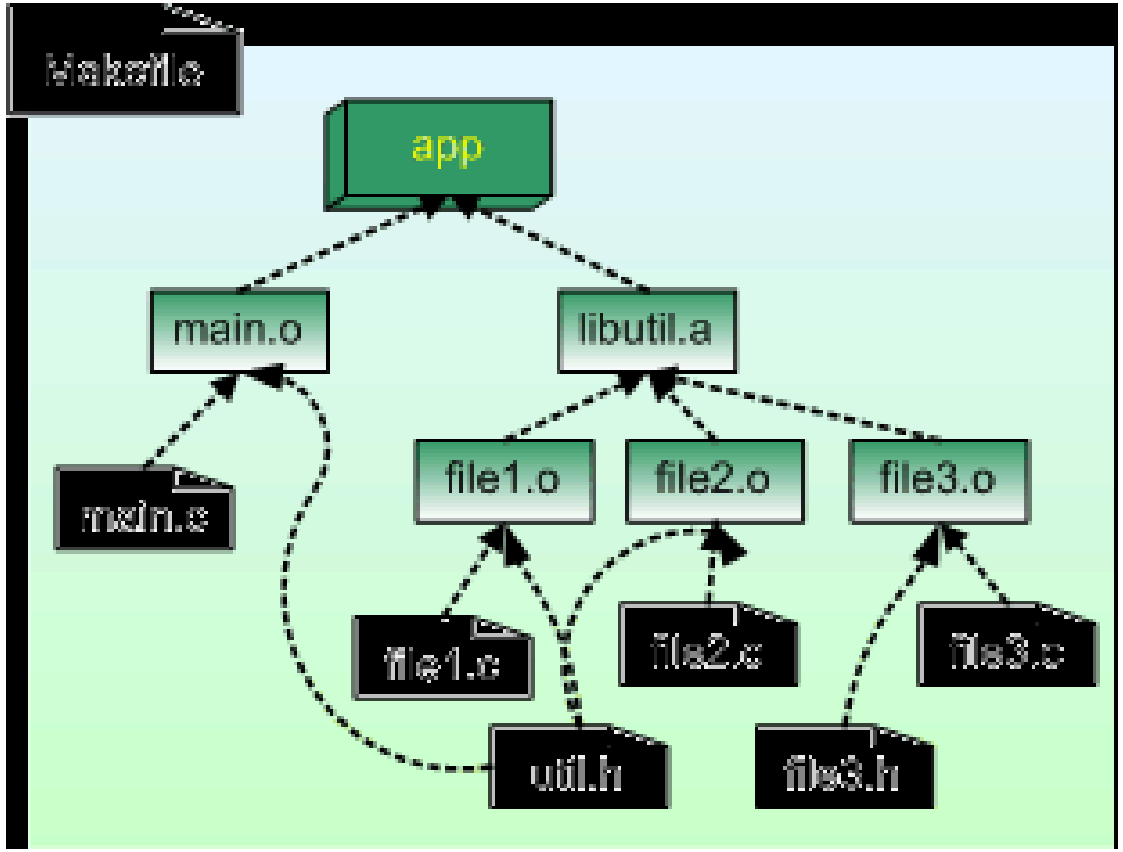
Type	Unix examples
Editors	`vi`, `emacs`
Pretty printers	`cb`, `indent`
Pre-processors (esp. macros)	`cpp`, `m4`, `watfor`
Debuggers	`adb`, `sdb`, `dbx`, `gdb`
Style checkers	`lint`, `purify`
Module management	`make`
Version management	`sccs`, `rcs`
Assemblers	`as`
Link editors, loaders	`ld`, `ld-so`
Perusal tools	`More`, `less`, `od`, `nm`
Program cross-reference	`ctags`

Source Code Editor
Notepad, Notepad++





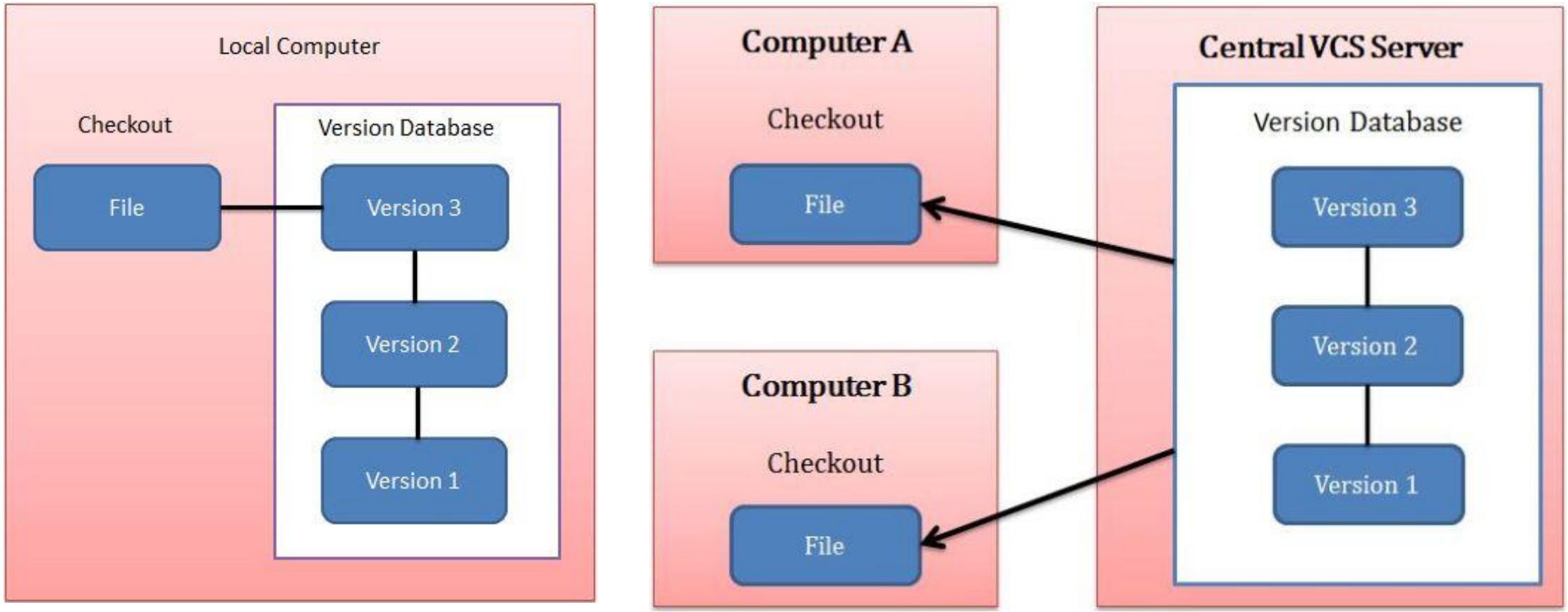
make: Program Builder




```
# define default target (first target = default)
# it depends on 'hello.o' (which will be created if necessary)
# and hello_func.o (same as hello.o)
hello: hello.o hello_func.o
        gcc -Wall hello.o hello_func.o -o hello
# define hello.o target
# it depends on hello.c (and is created from it)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
        gcc -Wall -c hello.c -o hello.o
# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
        gcc -Wall -c hello_func.c -o hello_func.o
#
```

Version Control

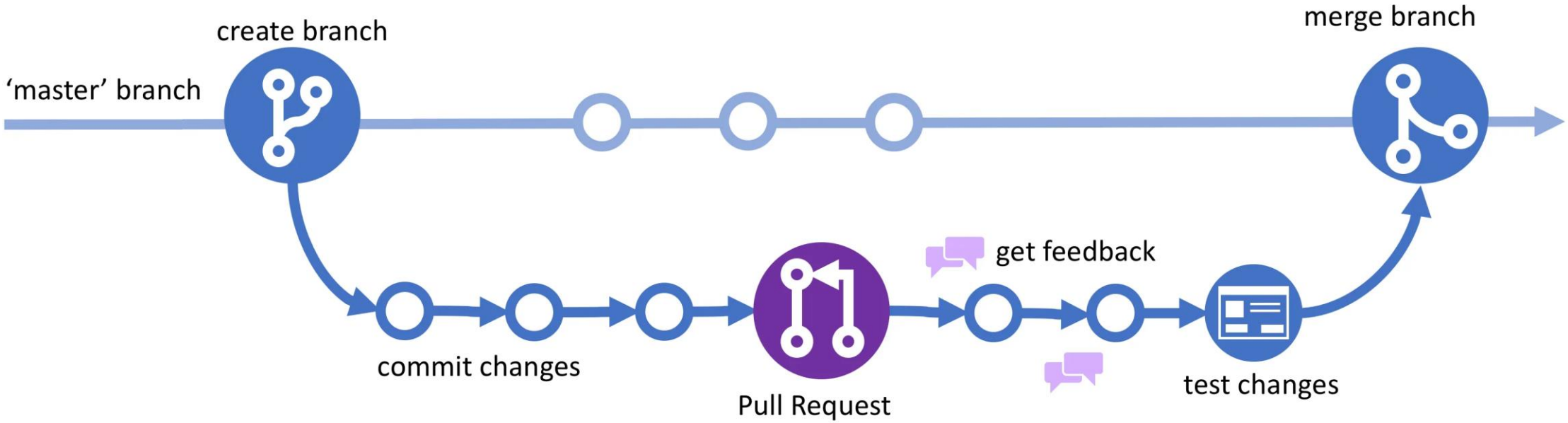
GNU RCS Revision Control System VS Github



Integrated Development Environment

Multi-Language IDEs	Mac	Linux	PC	C/C++	PHP	Python	Java	Ruby	HTML	Perl	.NET	CSS	JavaScript	Price
Eclipse	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes*	Yes	Yes*	Yes*	Yes	FOSS
NetBeans	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	FOSS
Komodo	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	$295
Geany	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	FOSS
Aptana	Yes	Yes	Yes	No	Yes*	Yes*	No	Yes*	Yes	No	No	Yes	Yes	FOSS
BlackAdder	No	Yes	Yes	No	No	Yes	No	Yes	No	No	No	No	No	$60

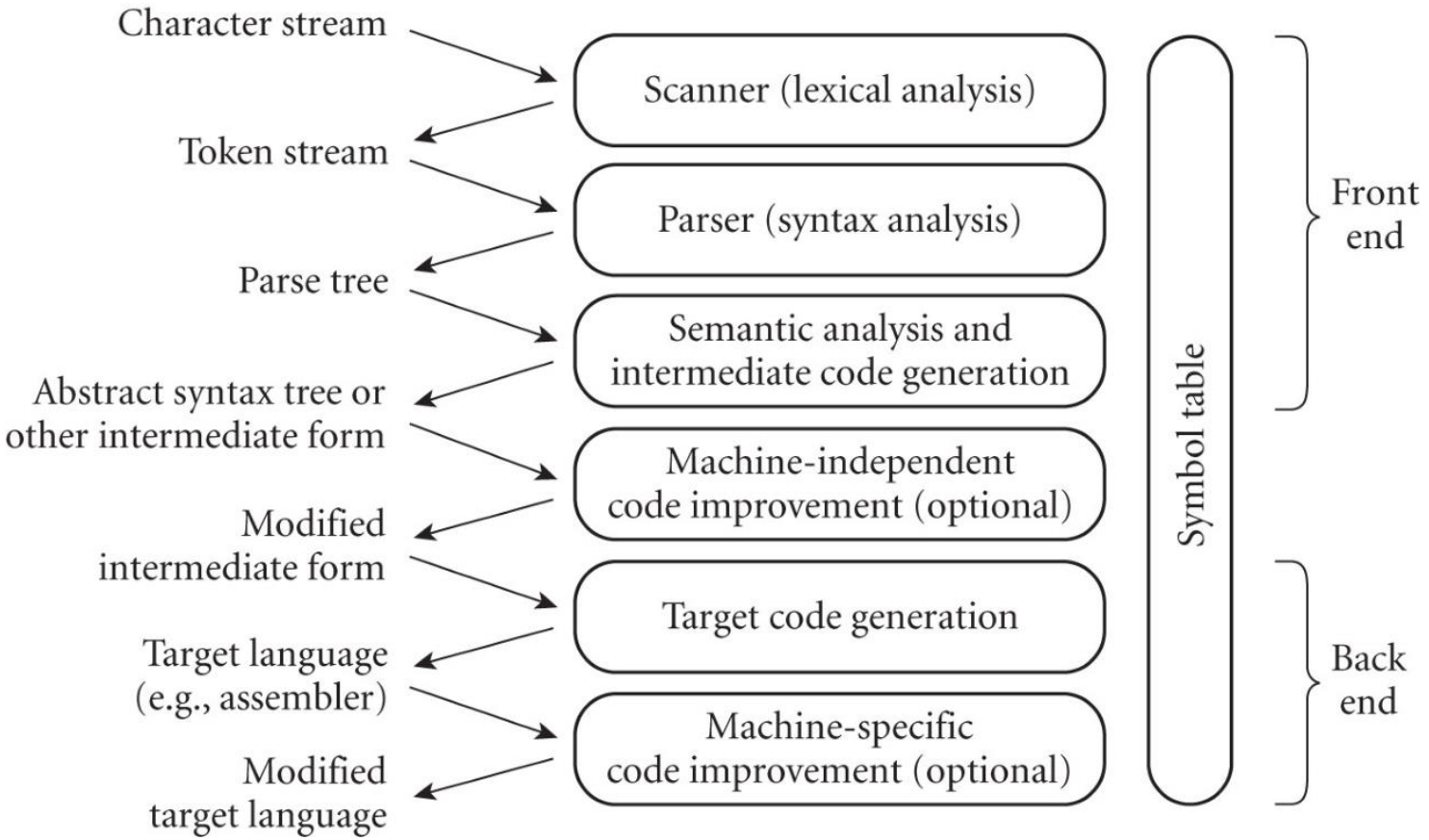
GitHub Flow

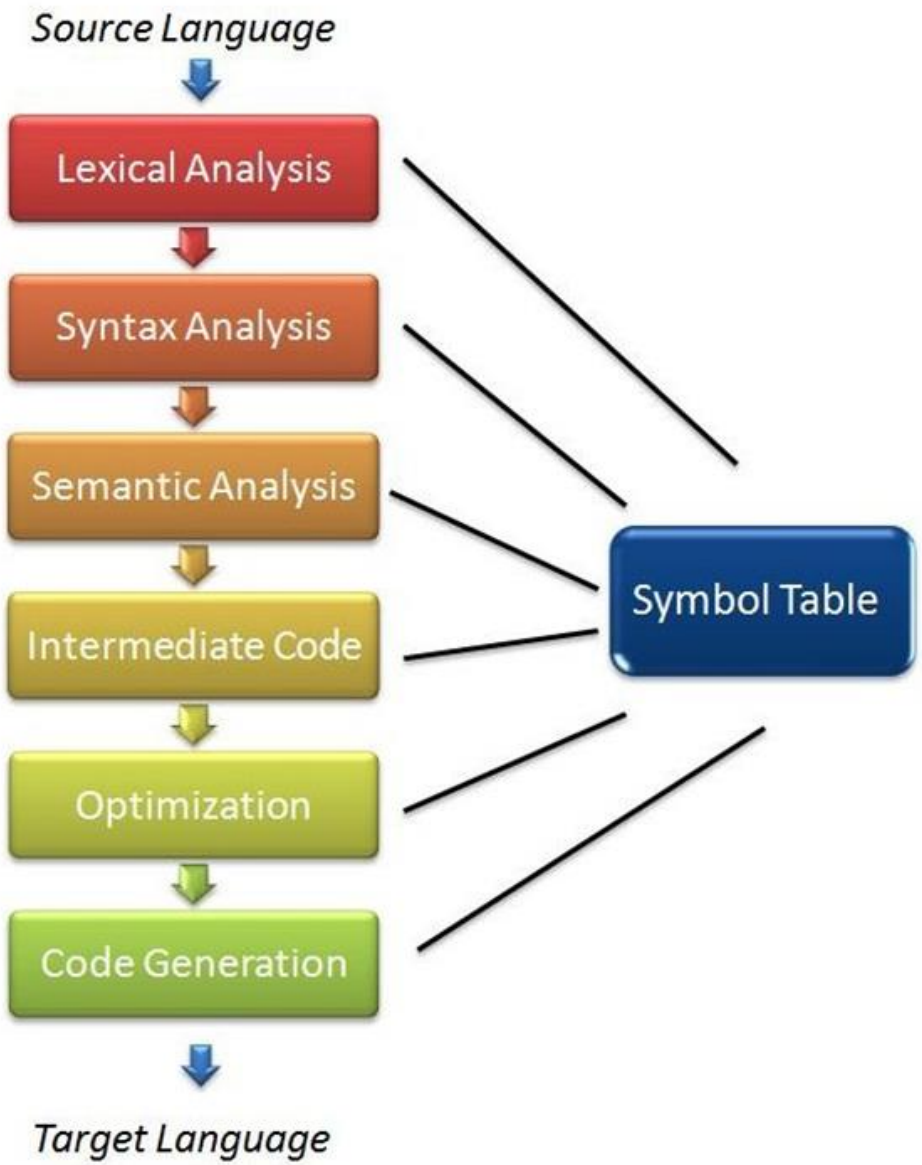


An Overview of Compiler Design I (6 phases)

SECTION 9

Phases of Compilation




Source Language
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code
Optimization
Code Generation
Target Language
Symbol Table

Scanning

- divides the program into "**tokens**", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
- we can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
- you can design a parser to take characters instead of tokens as input, but it isn't pretty
- scanning is recognition of a regular language, e.g., via **DFA**

DFA: Deterministic Finite Automaton

Deterministic Finite Automaton

Non-Random Finite State Machine

In Computer Science, a deterministic finite automaton (**DFA**)—also known as a deterministic finite accepter (**DFA**) and a deterministic finite state machine (**DFSM**)—is a finite-state machine that accepts and rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string.

The following example is of a DFA *M*, with a binary alphabet, which requires that the input contains an even number of 0s.

M = (*Q*, Σ, δ, q_0, *F*) where

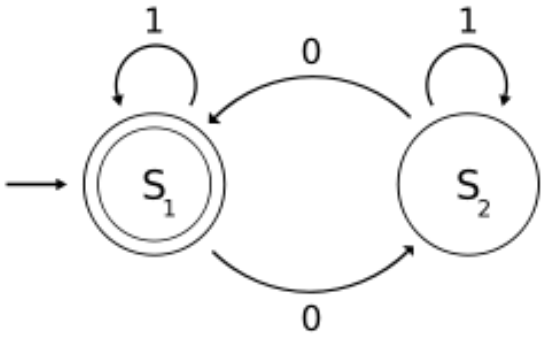
- *Q* = {S_1, S_2},
- Σ = {0, 1},
- q_0 = S_1,
- *F* = {S_1}, and
- δ is defined by the following state transition table:

Input

State

State Transition

	0	1
S_1	S_2	S_1
S_2	S_1	S_2



Parsing is recognition of a context-free language, e.g., via **PDA**

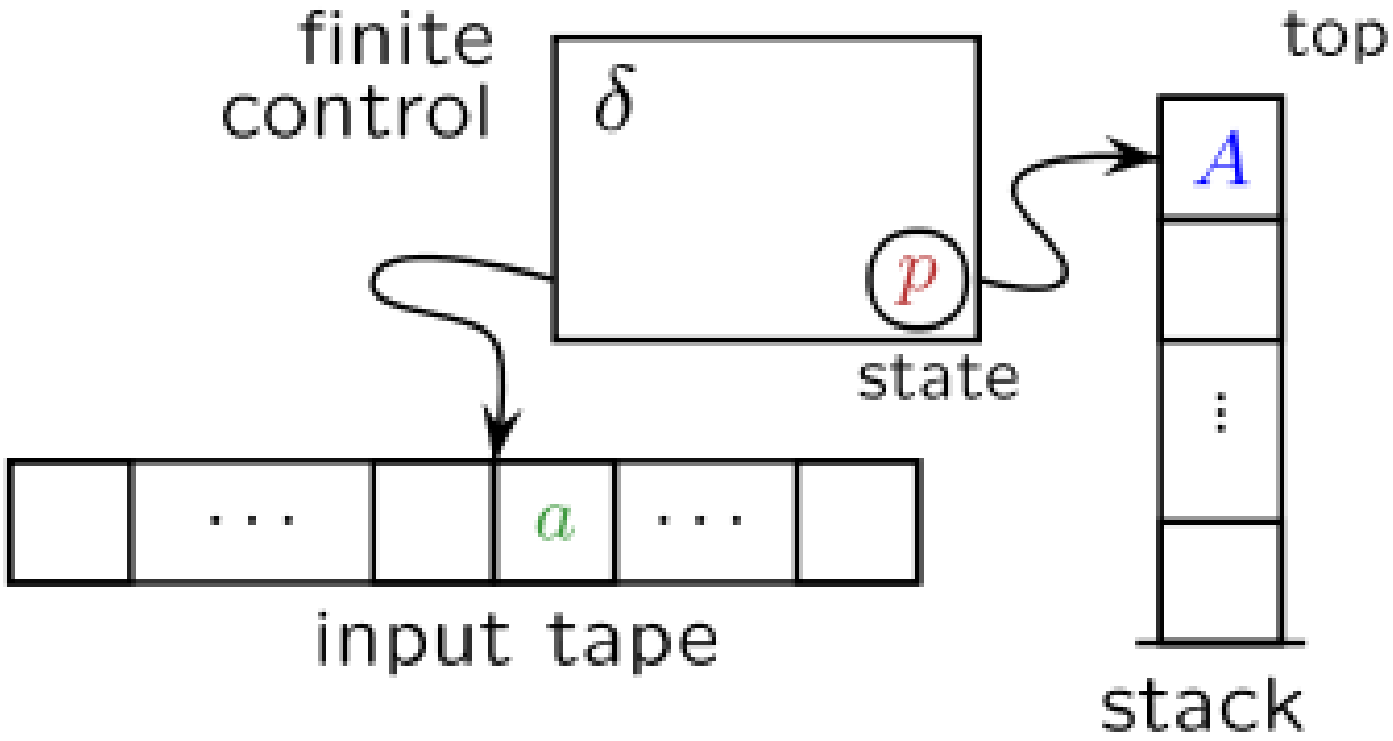
- Parsing discovers the "context free" structure of the program
- Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)

Syntax Analysis

PDA: Push Down Automaton

Push Down Automaton

In computer science, a pushdown automaton (PDA) is a type of automaton that employs a stack.



States are stored in stack.

Semantic analysis is the discovery of meaning in the program

The compiler actually does what is called **STATIC** semantic analysis. That's the meaning that can be figured out at compile time

Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's **DYNAMIC** semantics (Exception and Error)

Semantic Analysis

Intermediate form (IF) done after semantic analysis (if the program passes all checks)

- IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
- They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
- Many compilers actually move the code through more than one IF

Semantic Analysis

Last Two Phases

- **Optimization** takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - The term is a misnomer; we just improve code
  - The optimization phase is optional
- **Code generation phase** produces assembly language or (sometime) relocatable machine language

Code Optimization

Code Generation

Machine-specific Optimization and Symbol Table

- Certain **machine-specific optimizations** (use of special instructions or addressing modes, etc.) may be performed during or after **target code generation**
- **Symbol table:** all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
- This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

Code Generation

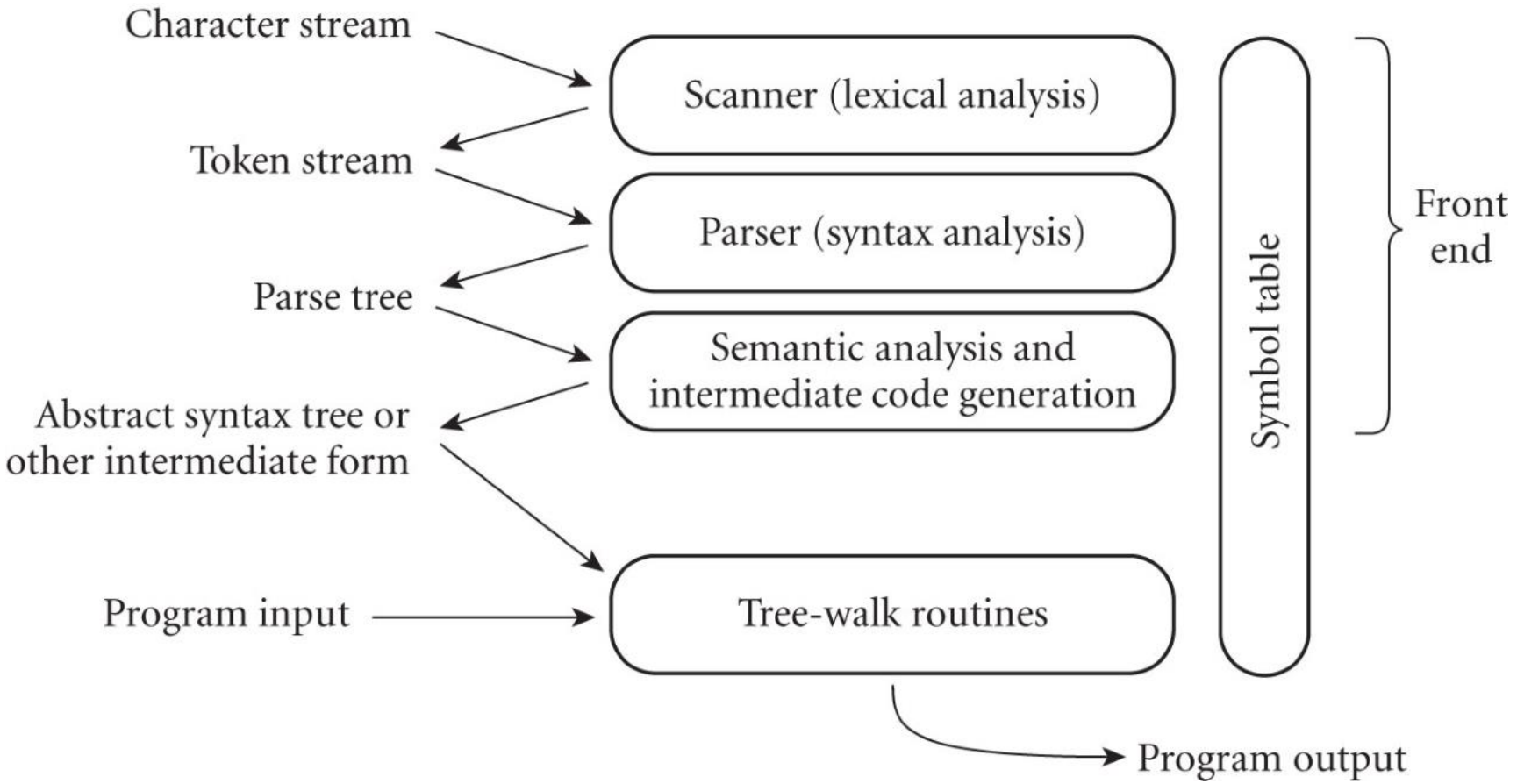
Symbol Table

An Overview of Compiler Design II

(Analysis in Front-End Compiler)

SECTION 10

Front-End Compiler



Lexical and Syntax Analysis

GCD Program (in C)

```
int main() {
 int i = getint(), j = getint();
 while (i != j) {
  if (i > j) i = i - j;
  else j = j - i;
 }
 putint(i);
}
```

Lexical and Syntax Analysis

- GCD Program Tokens
  - Scanning (lexical analysis) and parsing recognize the structure of the program, groups characters into tokens, the smallest meaningful units of the program

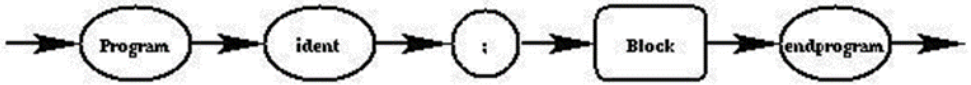
```
int     main   (   )       {
int     i      =   getint   (   )   ,   j   =   getint   (   )   ;
while   (      i   !=       j   )   {
if      (      i   >        j   )   i   =   i   -       j   ;
else    j      =   j        -   i   ;
}
putint  (      i   )        ;
}
```

Lexical and Syntax Analysis

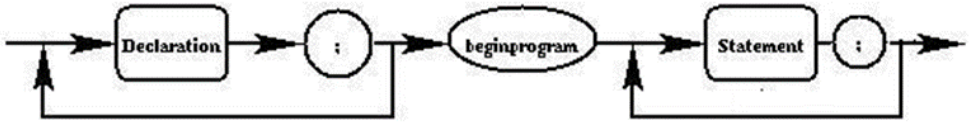
- Context-Free Grammar and Parsing
  - Parsing organizes tokens into a parse tree that represents higher-level constructs in terms of their constituents
  - Potentially recursive rules known as context-free grammar define the ways in which these constituents combine

Syntax Diagram

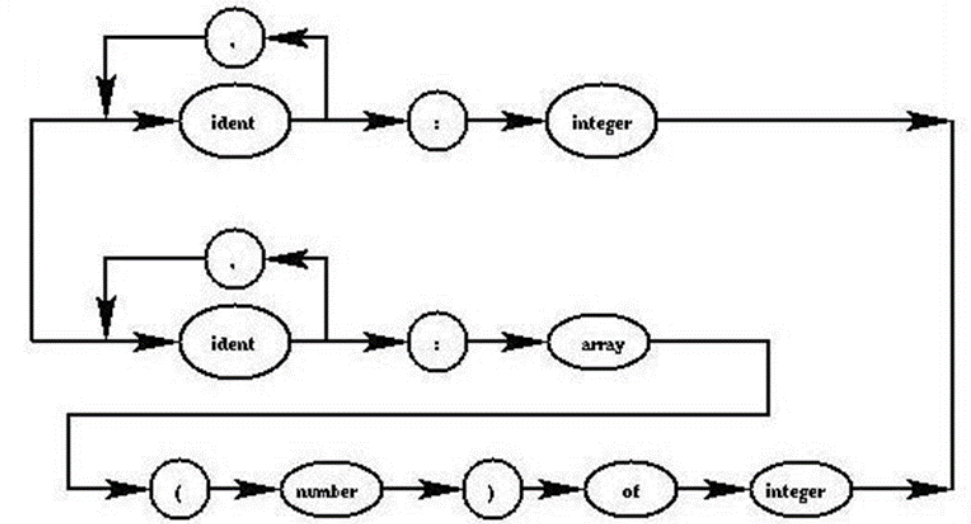
Program:



Block:

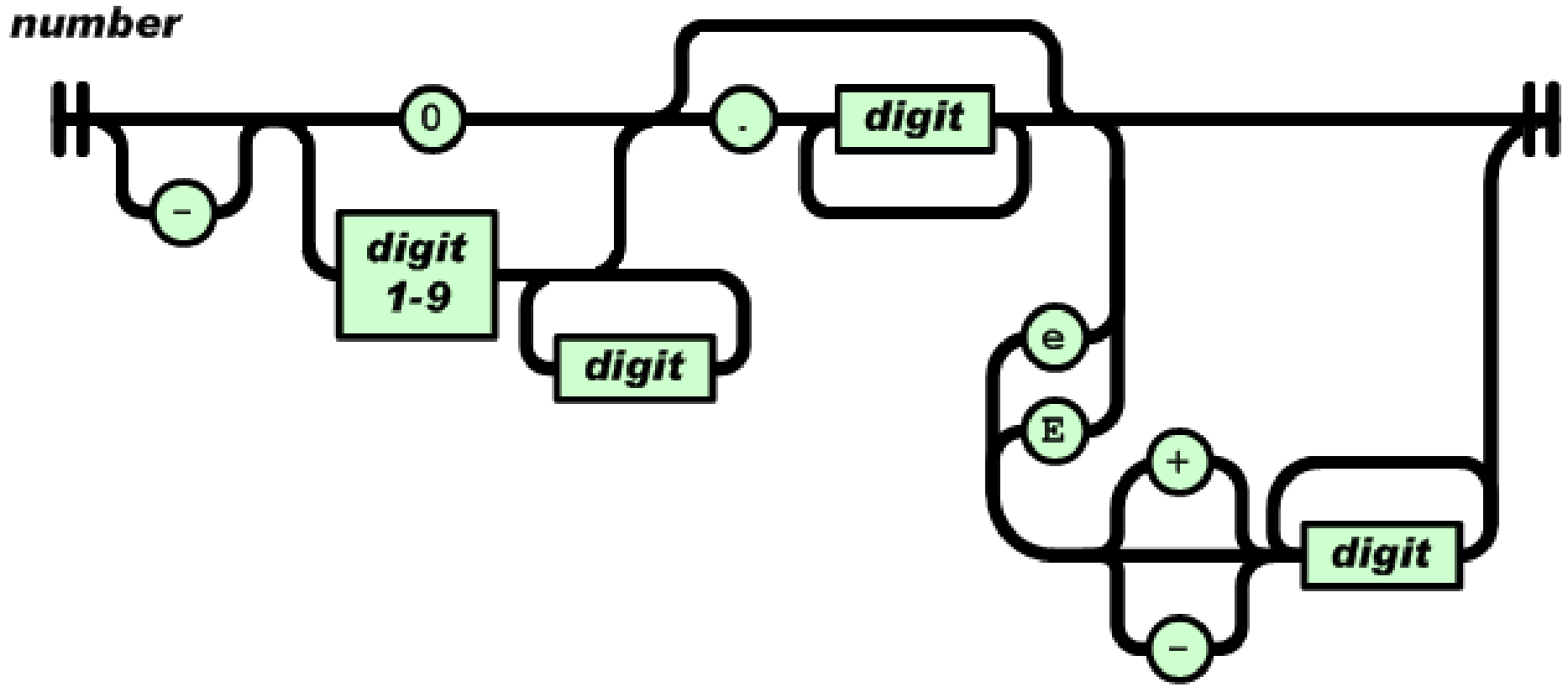


Declaration



Context-Free Grammar

BNF = Backus Normal Form or Backus Naur Form



Context-Free Grammar and Parsing

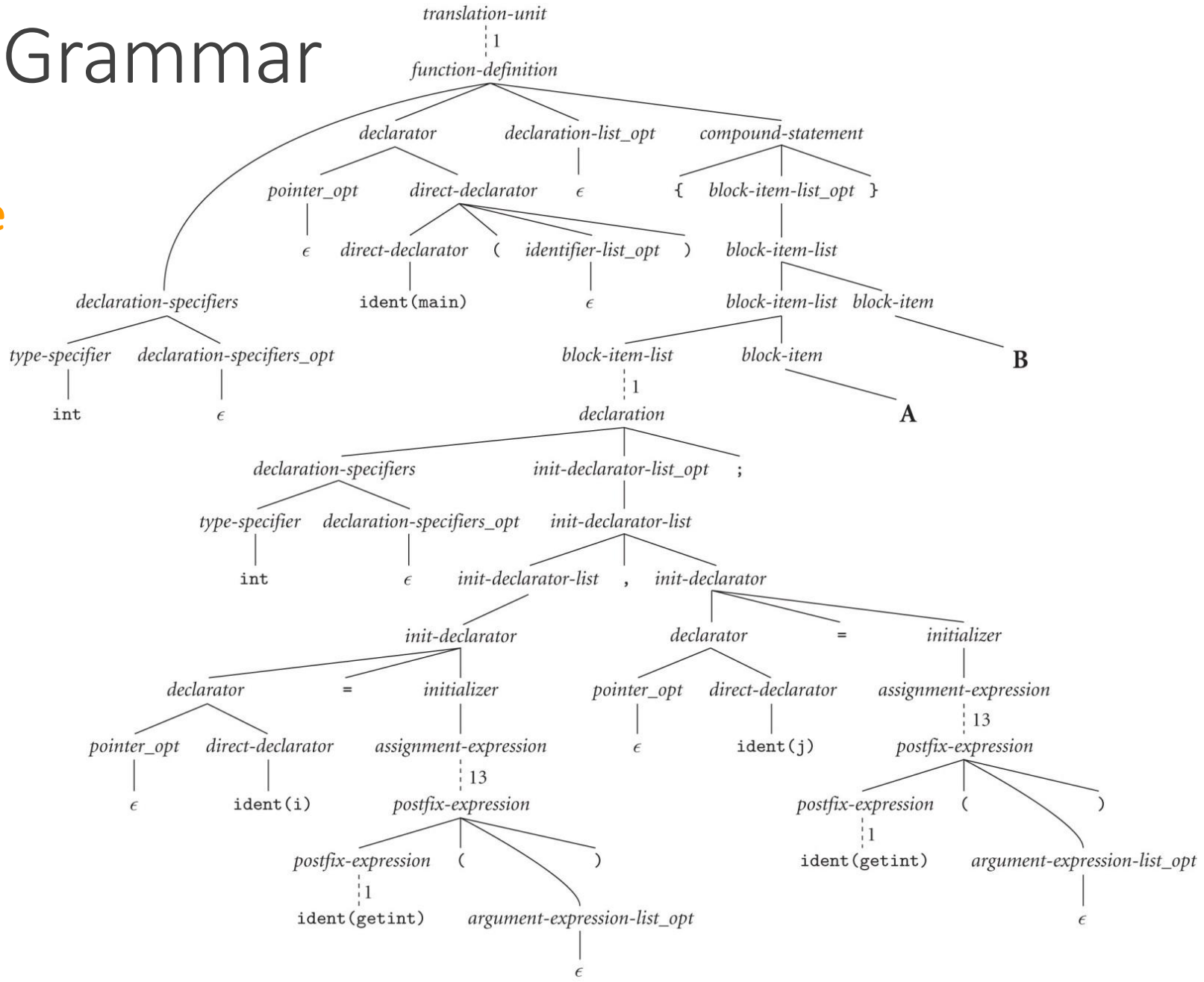
Example (while loop in C)

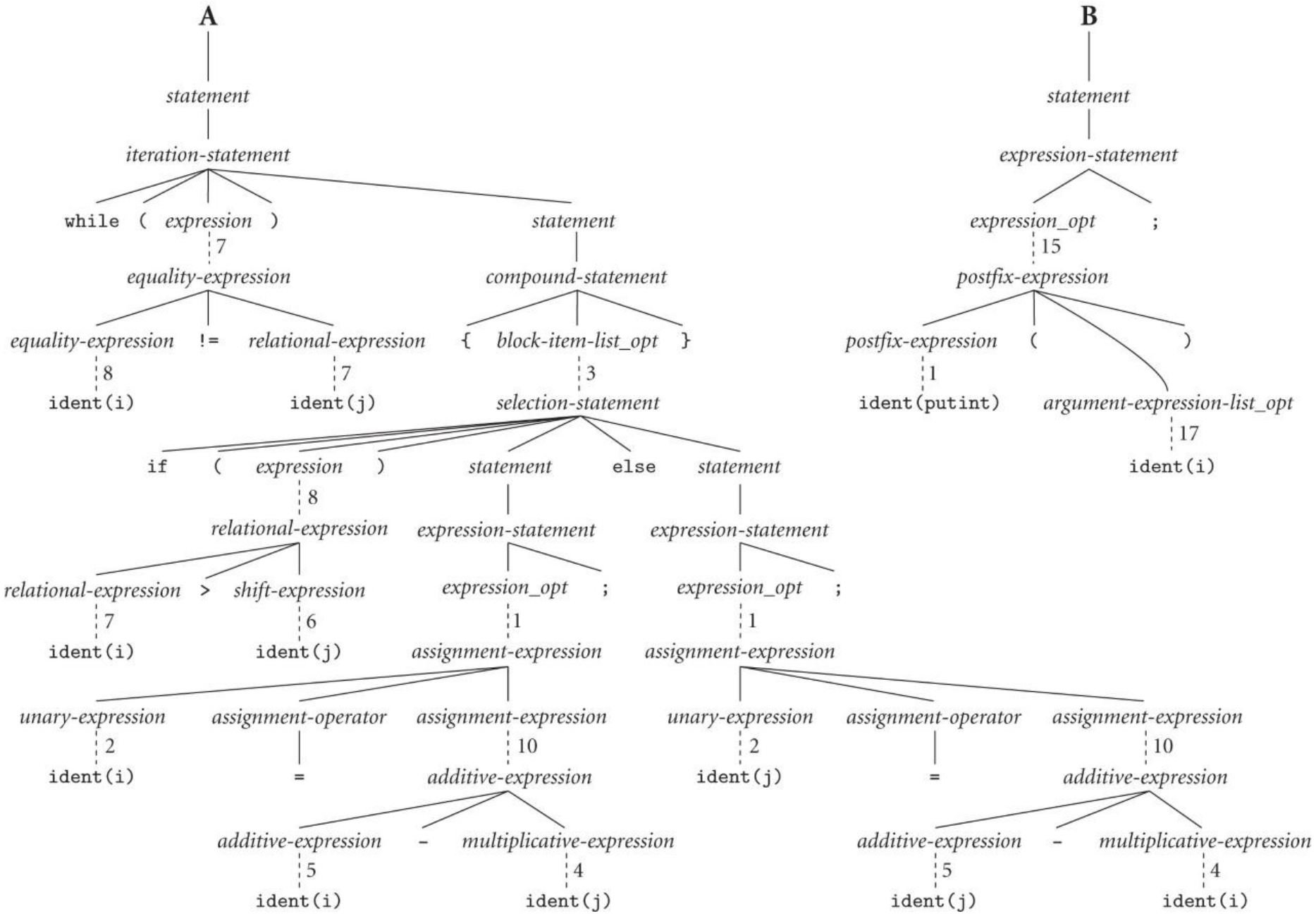
iteration-statement → *while (expression) statement*

statement, in turn, is often a list enclosed in braces:
statement → *compound-statement*
compound-statement → *{ block-item-list opt }*
where
block-item-list opt → *block-item-list*
or
block-item-list opt → ϵ
and
block-item-list → *block-item*
block-item-list → *block-item-list block-item*
block-item → *declaration*
block-item → *statement*

Context-Free Grammar and Parsing

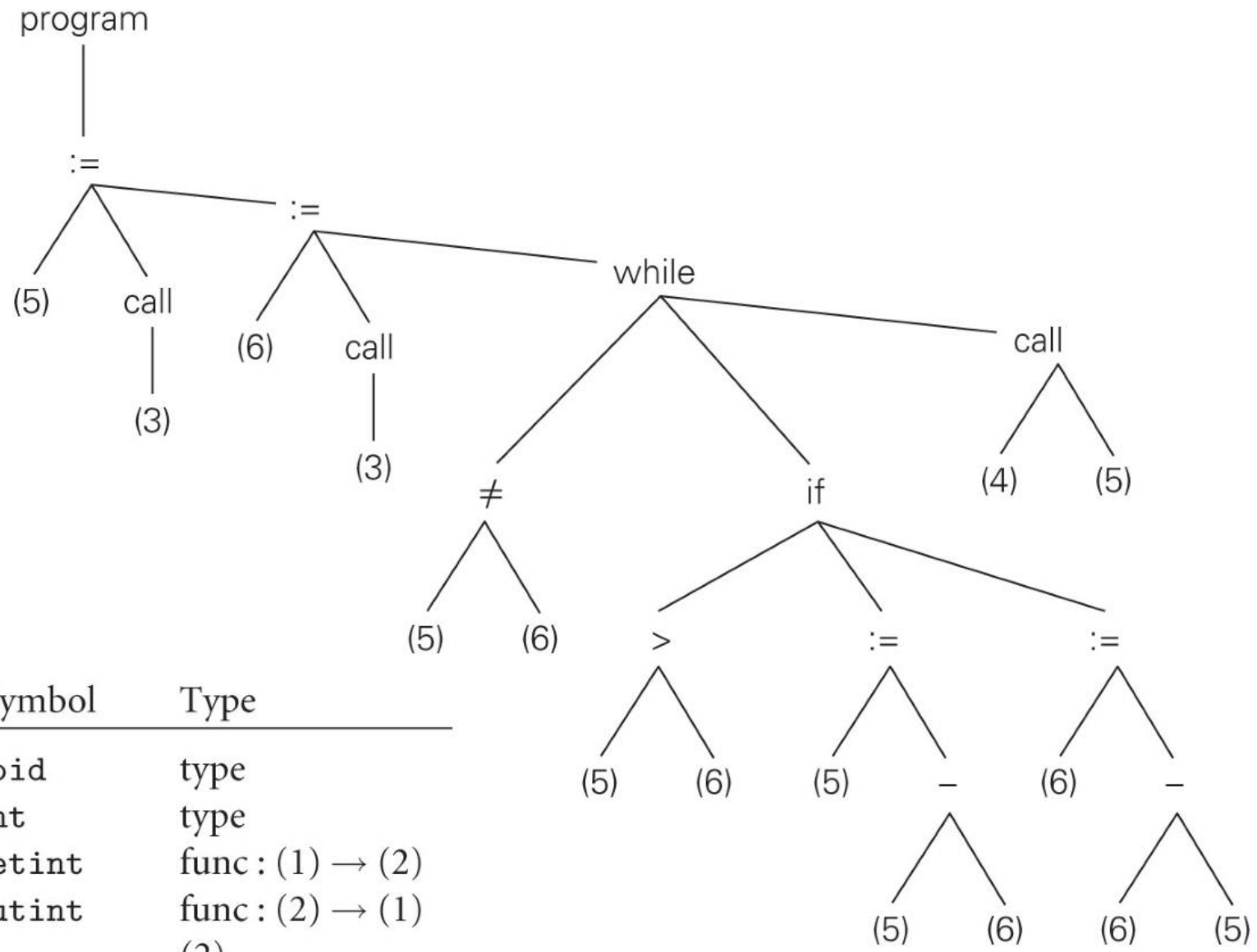
GCD Program Parse Tree




A
statement
iteration-statement
while (expression)
7
equality-expression
equality-expression != relational-expression
8
ident(i)
7
ident(j)
statement
compound-statement
{ block-item-list_opt }
3
selection-statement
if (expression) statement else statement
8
relational-expression
relational-expression > shift-expression
7
ident(i)
6
ident(j)
expression-statement
expression_opt ;
1
assignment-expression
unary-expression assignment-operator assignment-expression
2
ident(i)
=
10
additive-expression
additive-expression - multiplicative-expression
5
ident(i)
4
ident(j)
expression-statement
expression_opt ;
1
assignment-expression
unary-expression assignment-operator assignment-expression
2
ident(j)
=
10
additive-expression
additive-expression - multiplicative-expression
5
ident(j)
4
ident(i)
B
statement
expression-statement
expression_opt ;
15
postfix-expression
postfix-expression ( )
1
ident(putint)
argument-expression-list_opt
17
ident(i)

Syntax Tree

GCD Program Parse Tree



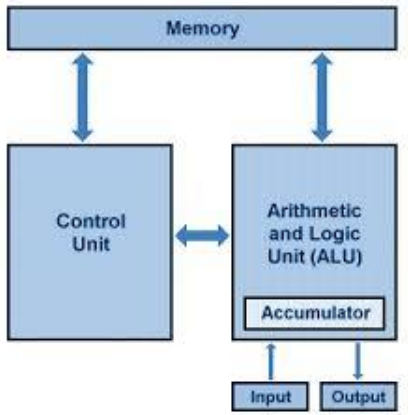
Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

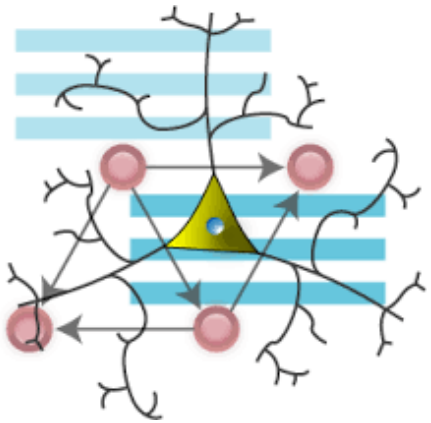
The Development of a Programming Language

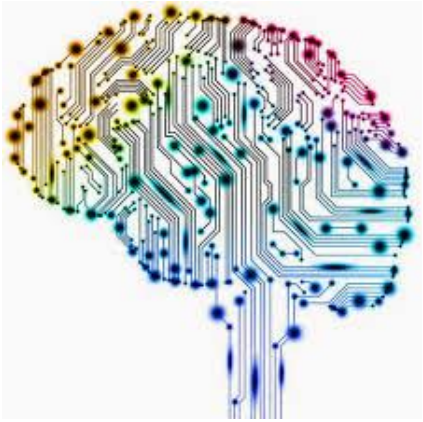
SECTION 11

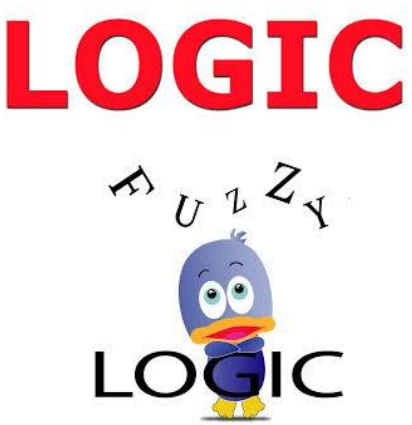
The Development of a Programming Language

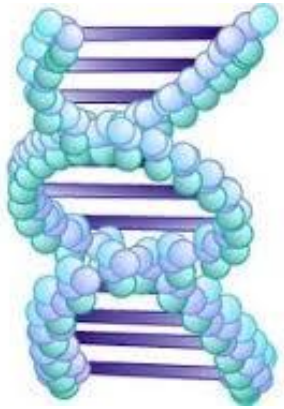
Computing Models

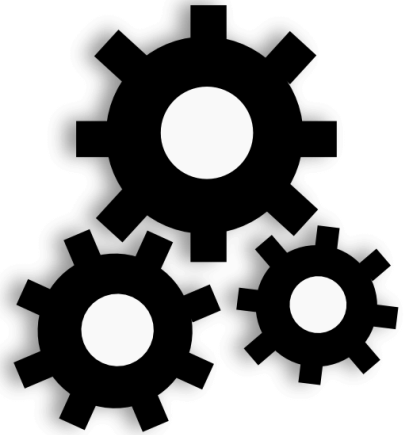




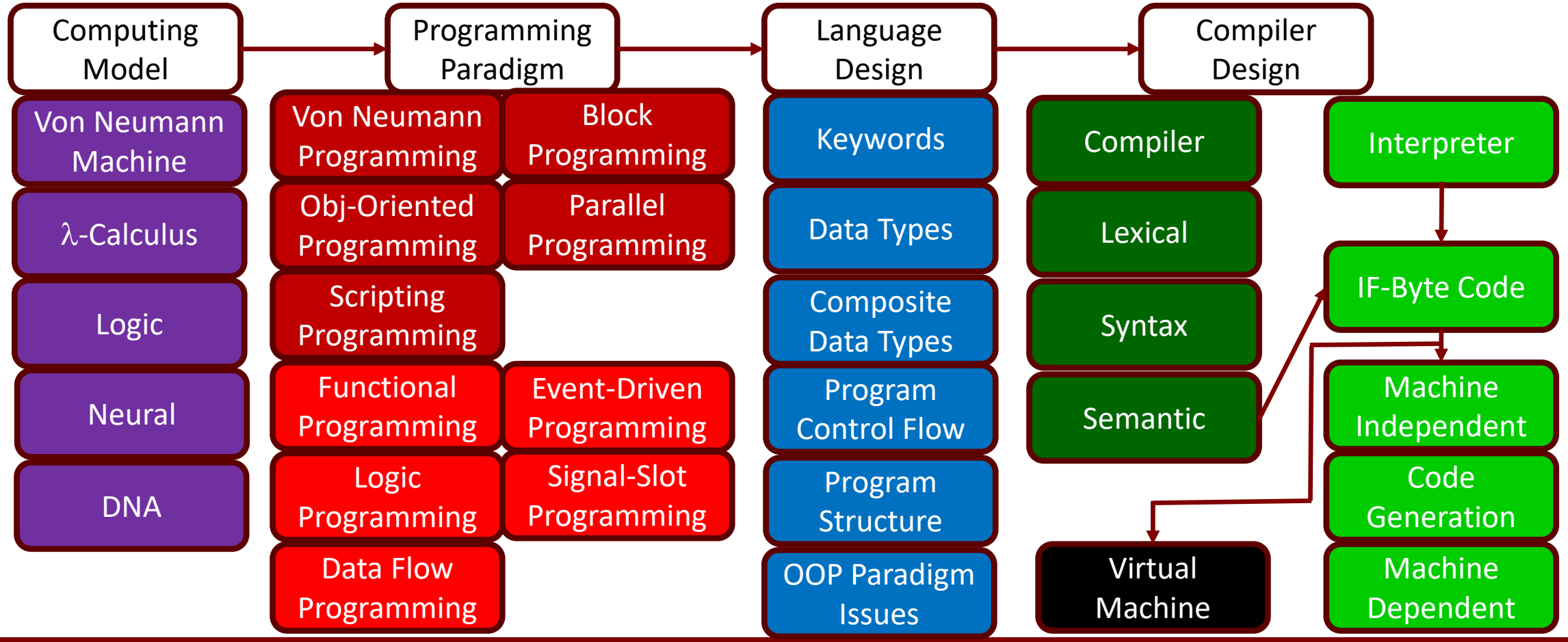






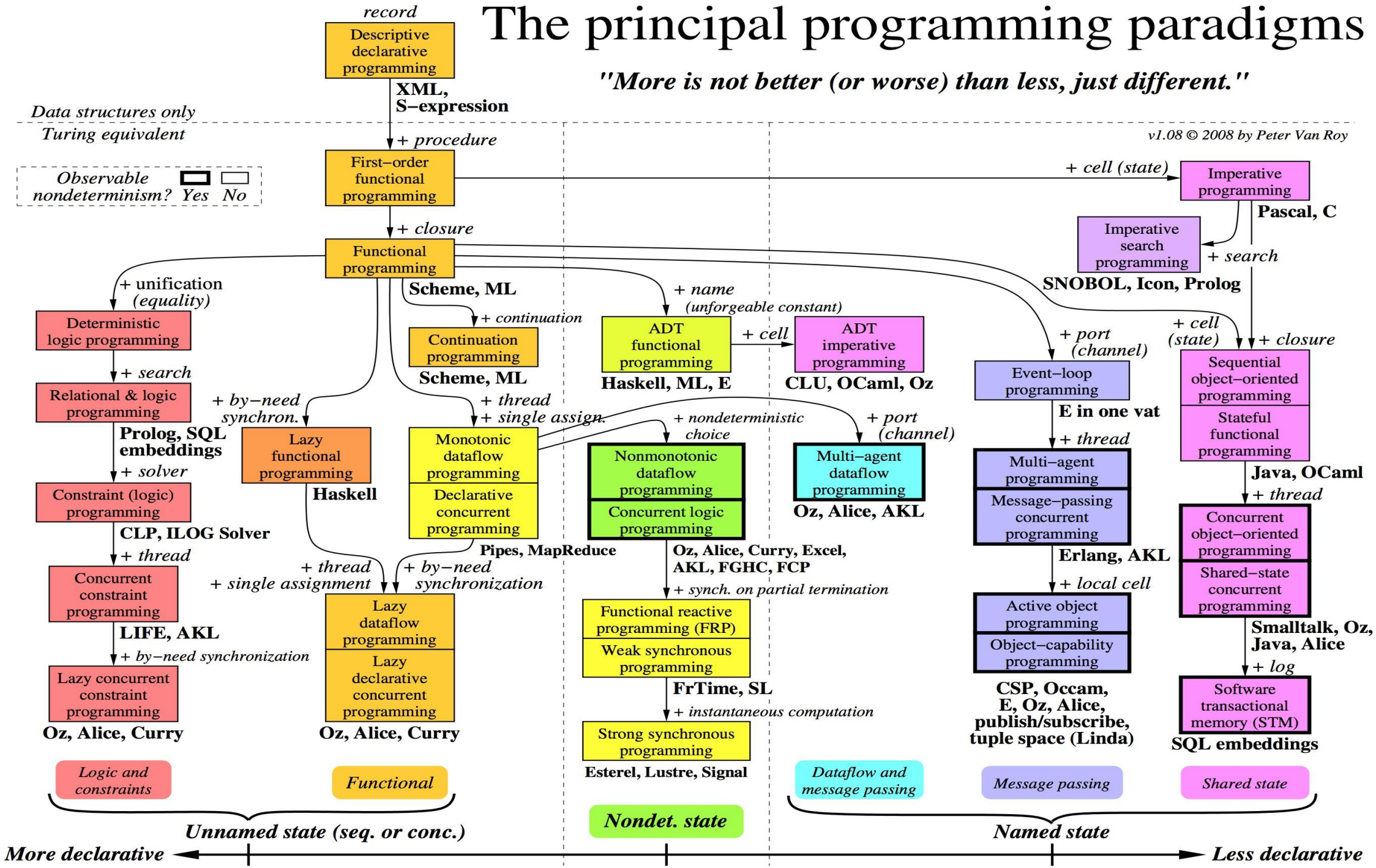


The Development of a Programming Language



The principal programming paradigms

"More is not better (or worse) than less, just different."



Run Time Environment

Programming Language Beyond Compiler

Programming Essentials
C-Equivalent

Obj-Oriented Programming
C++ Equivalent

GUI	Data Structure
File I/O	Concurrency
Exception	Generics
Numerical	Database
Geometry	Chart Plotting
String	Networking
Data Collection	Unit Test

Development Kit, APIs

Design Framework
Client-Server
Programming

Design Studio
Compilers
IDE
APIs
Framework
etc.