CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 1 Introduction

LECTURE 1: INTRODUCTION

DR. ERIC CHOU

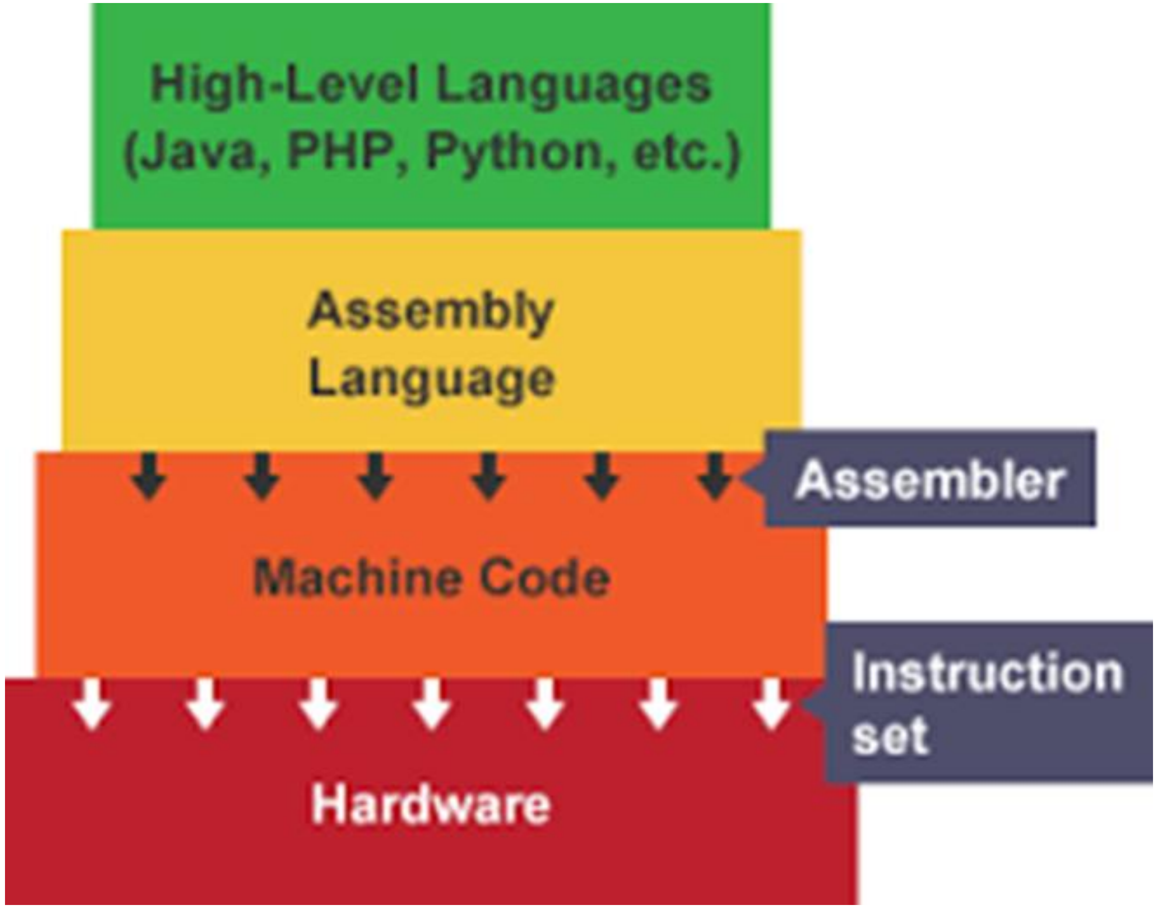
IEEE SENIOR MEMBER

Objectives

- Introduction to Programming Languages
- Programming Paradigms
- Interpretation of Computer Programs: 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

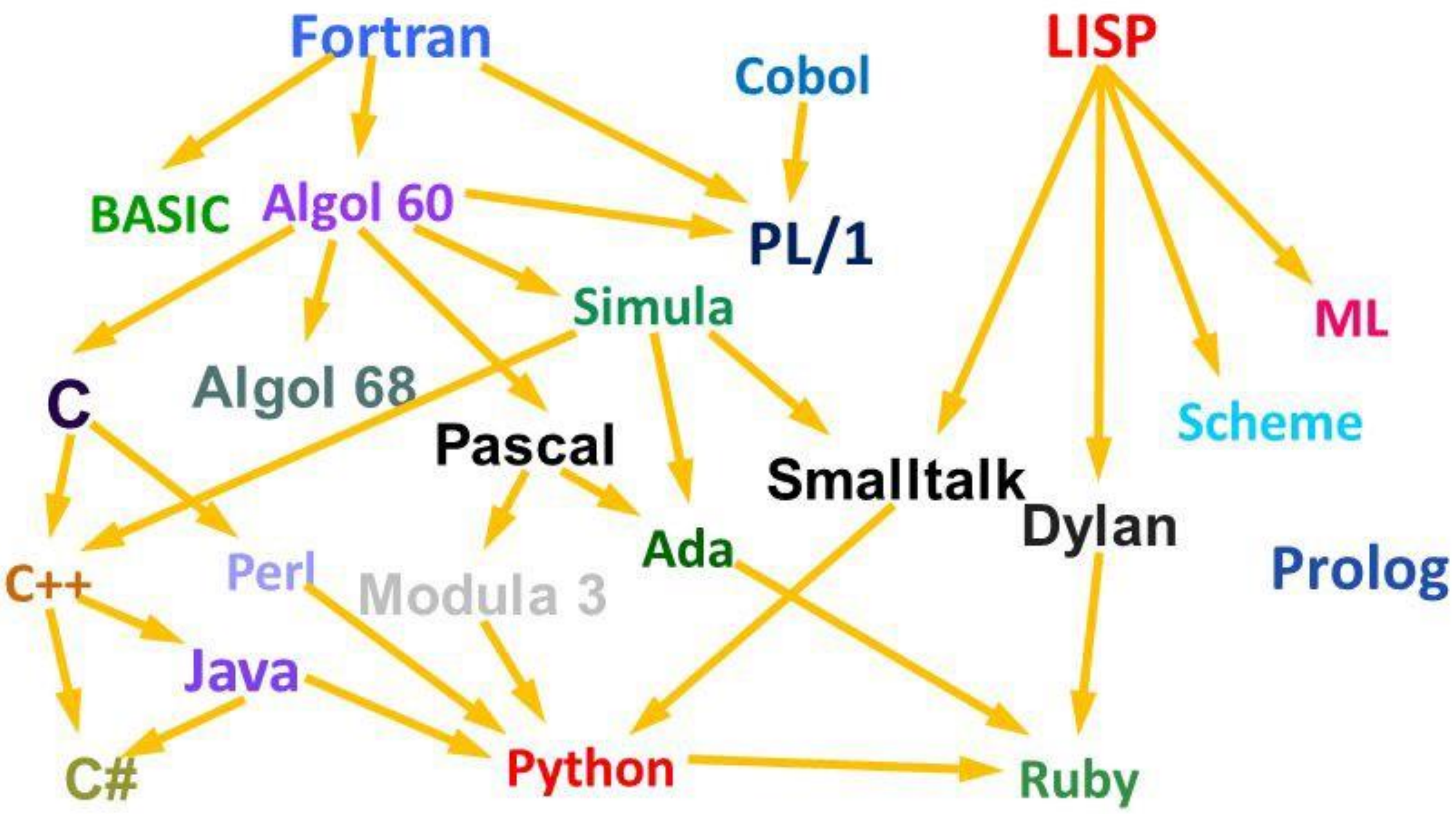
Assembly Language GCD Code

```
    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
```

Most Popular Programming Languages 2021

A family tree of languages

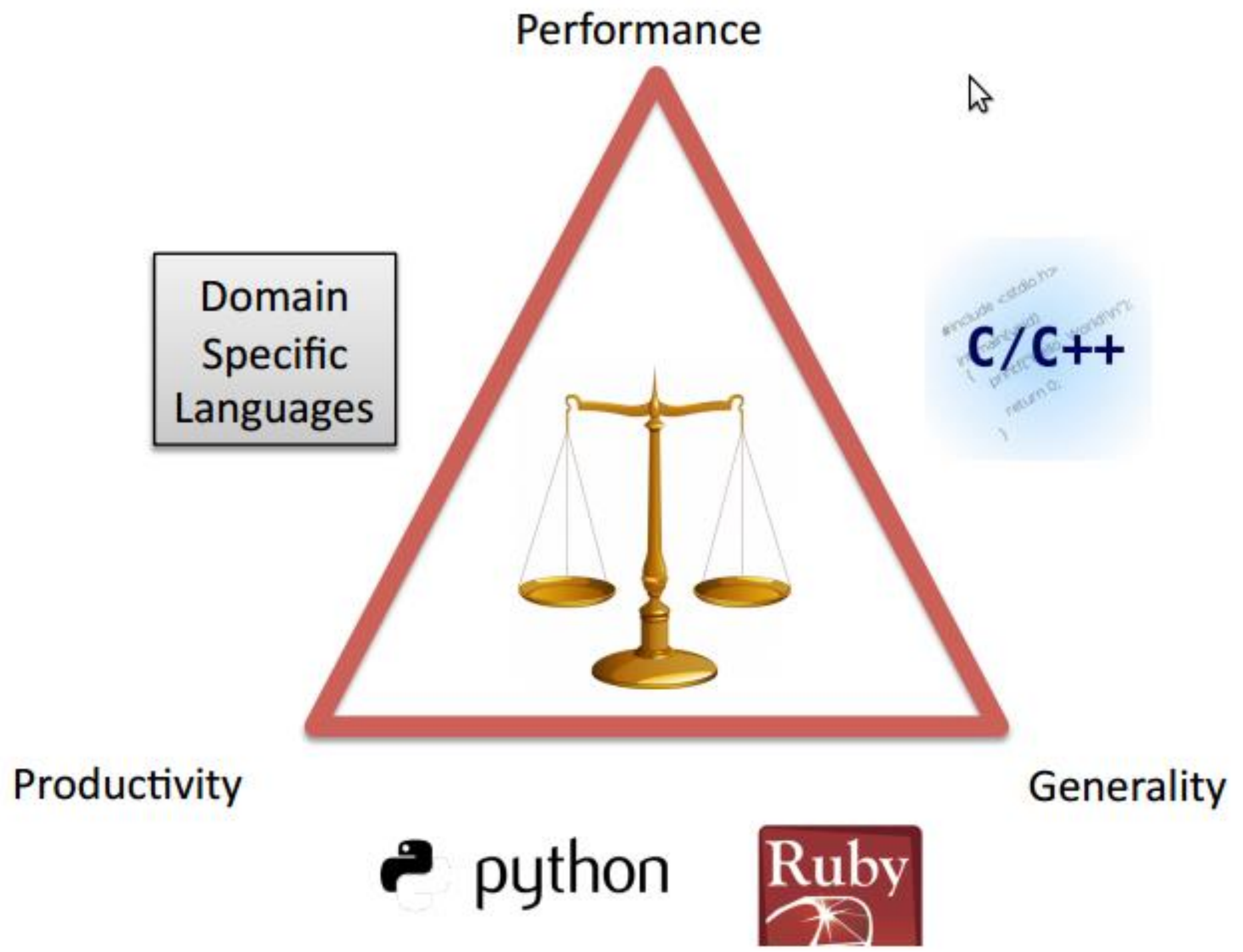
Some of the 2400 + programming languages



Why are there so many programming languages?

1. Evolution
2. Socio-Economic Reasons
3. Orientation toward special purposes
4. Orientation toward special hardware

EVOLUTION
COBOL
OBJ C
php
Java
perl
JS
CODING MORTALS
c
c++
java
python
go
dart
swift
Scala
GO
Kotlin
Dart
Swift


Performance
Domain Specific Languages
C/C++
Productivity
Generality
python
Ruby

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 implementator's point of view

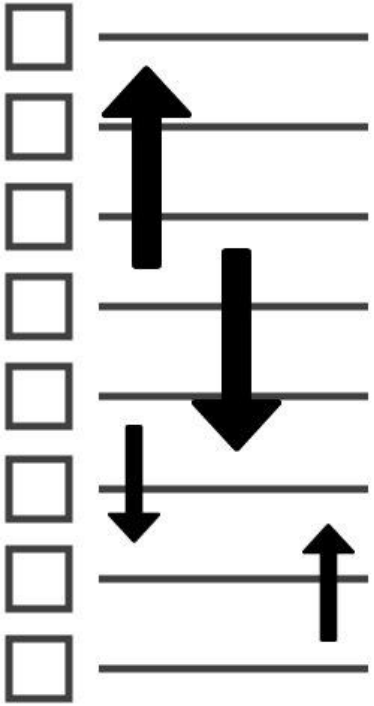
Programming Paradigms

SECTION 2

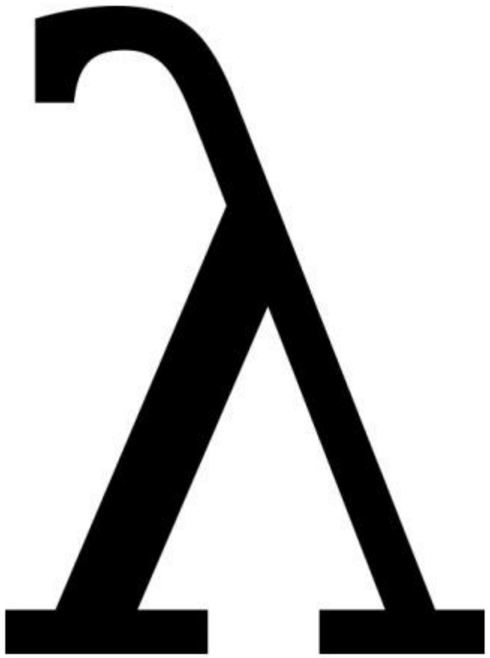
Computer Scientist Group Language as …

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, …

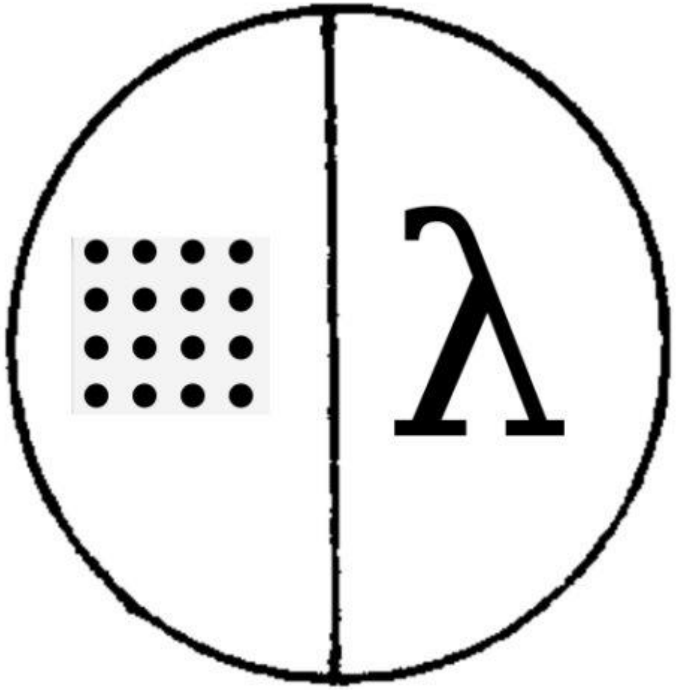
Imperative



Functional

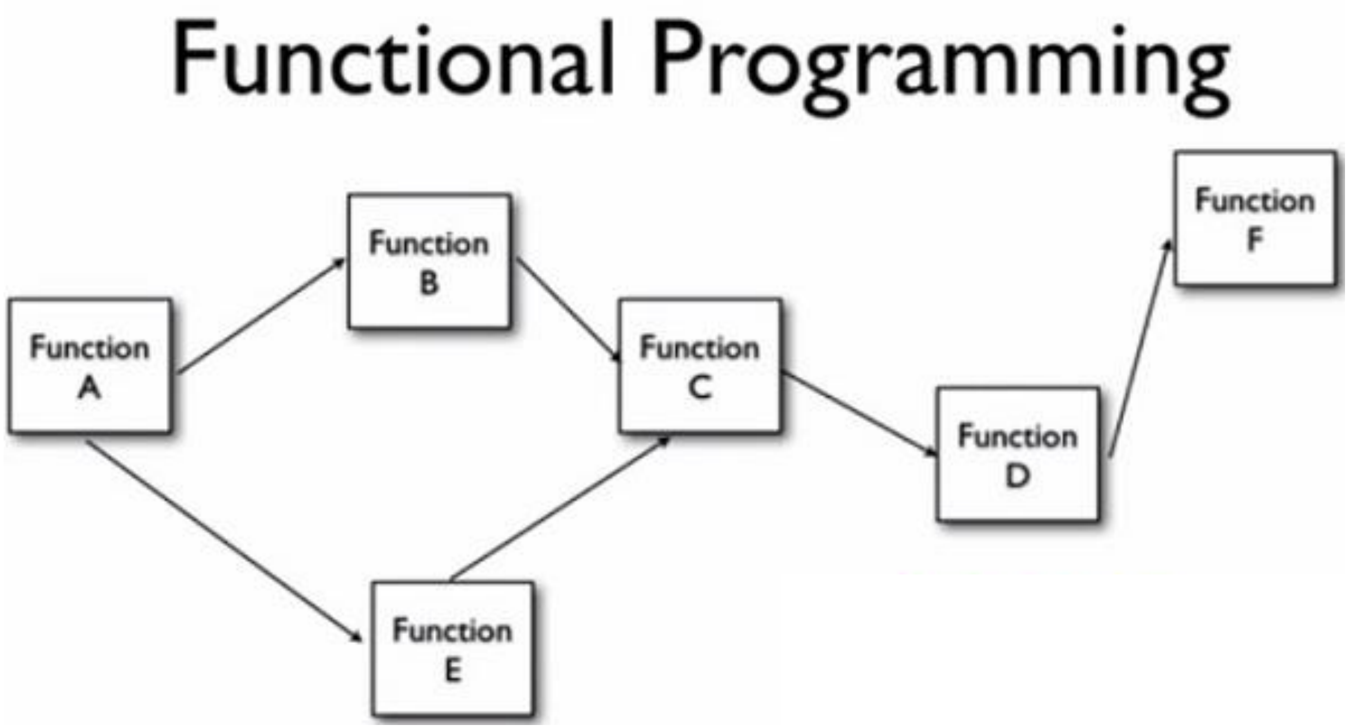


Object-Oriented



Programming Paradigm

Declarative Languages



Driven by Functional Call

Programming Paradigm

Declarative Languages

Logic Programming

Program Window

no space Arguments/Objects

Facts

```
likes(john, jane).
likes(jane, john).
likes(jack, jane).
```

dot necessary to end statements

Predicate name

Rule

variables

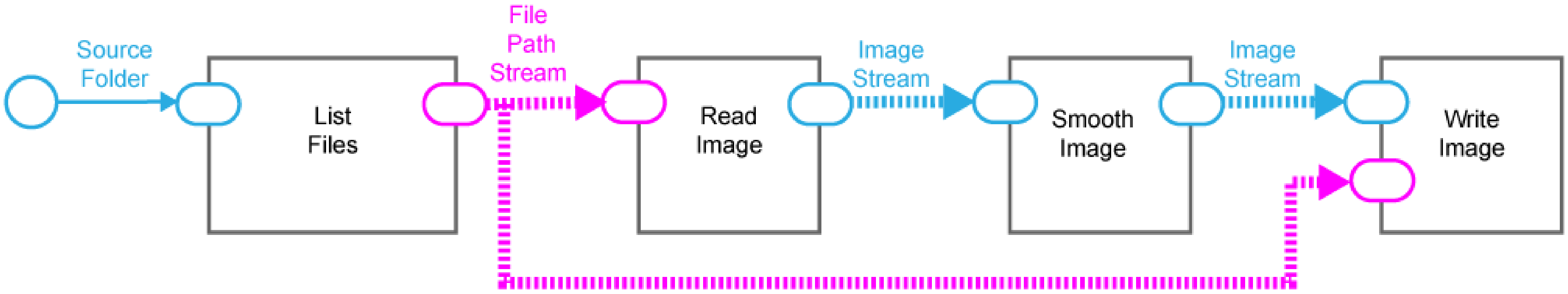
$\underbrace{\texttt{friends(X, Y)}}_{\text{head}}$:- $\underbrace{\texttt{likes(X, Y), likes(Y, X).}}_{\text{body}}$

Driven by Logic Reasoning

Programming Paradigm

Declarative Languages

Data Flow Programming



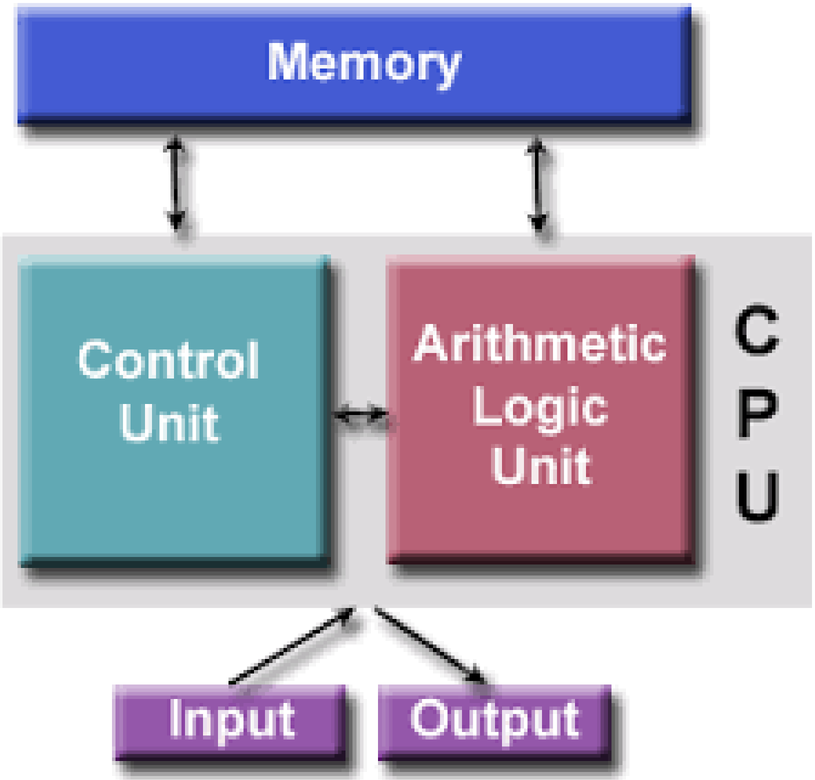
Driven by Data Flow

Programming Paradigm

Imperative Languages

Von Neumann Programming

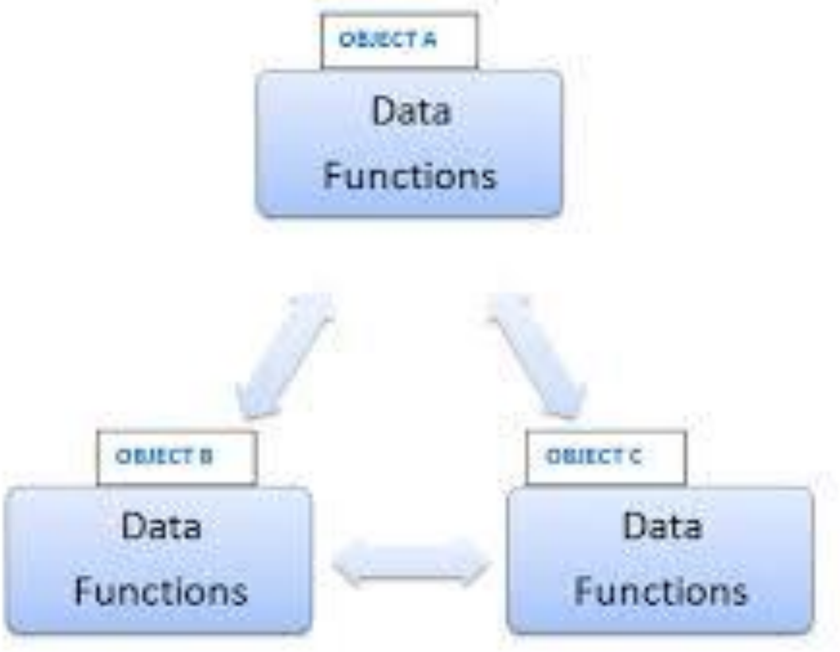
(Accumulator Model)

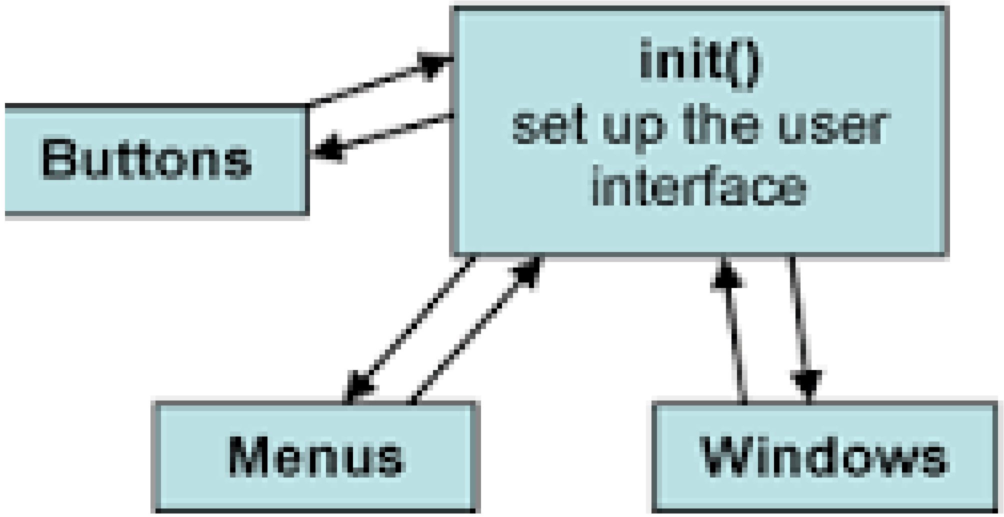


Programming Paradigm

Imperative Languages

Object-Oriented Programming



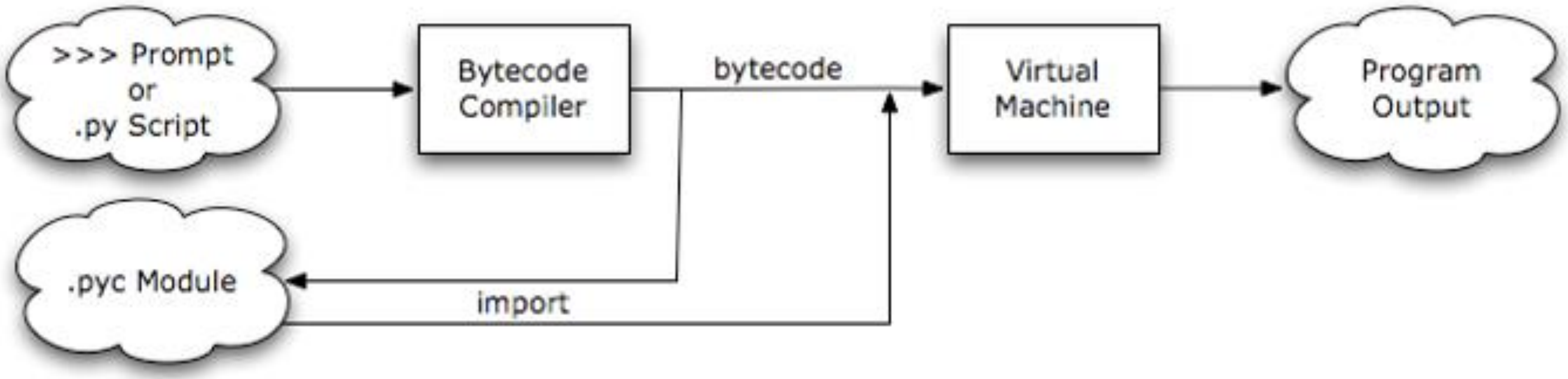


Event-Driven Programming

Programming Paradigm

Imperative Languages

Scripting Programming



Programming on Another Program (Virtual Machine)

```
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).
```

The Interpretation of Computer Programs

SECTION 3 INTERPRETATION VERSUS COMPILATION

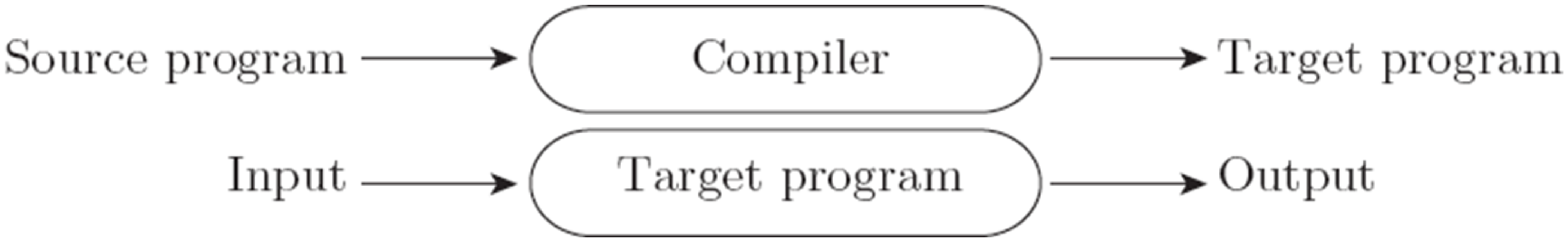
Purpose of Compilation and Interpretation

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

Pure Compilation

C -> Machine Code; C -> Java?

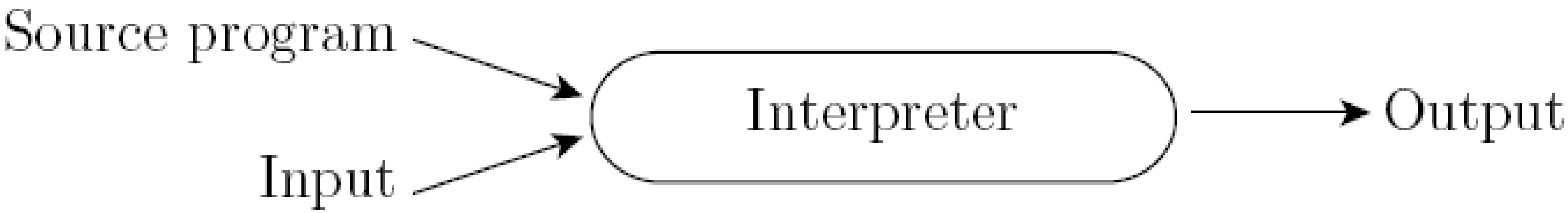
The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



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



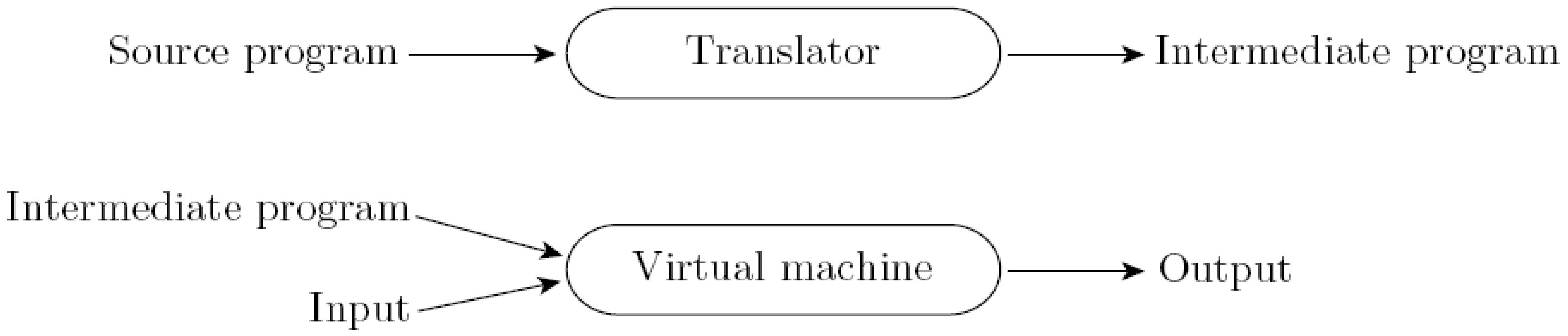
Comparison

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

- Compilation
  - Better performance

Hybrids

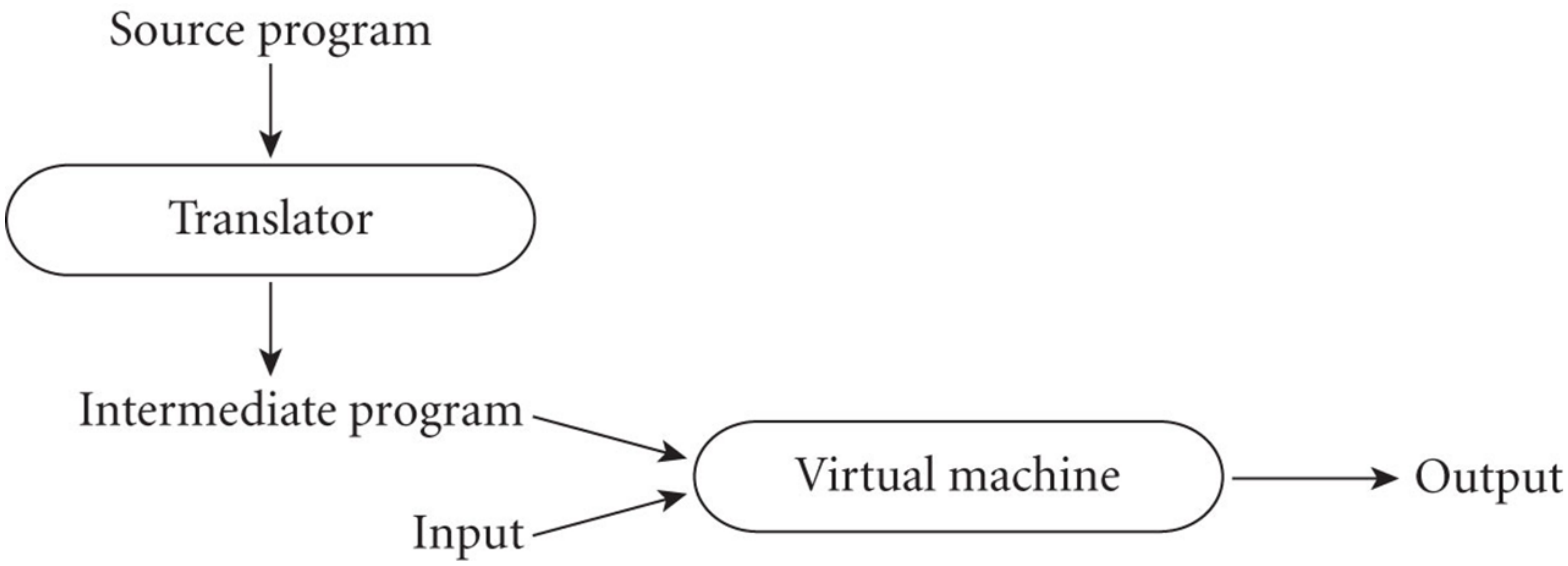
- Common case is compilation or simple pre-processing, followed by interpretation
- Most modern language implementations include a mixture of both compilation and interpretation

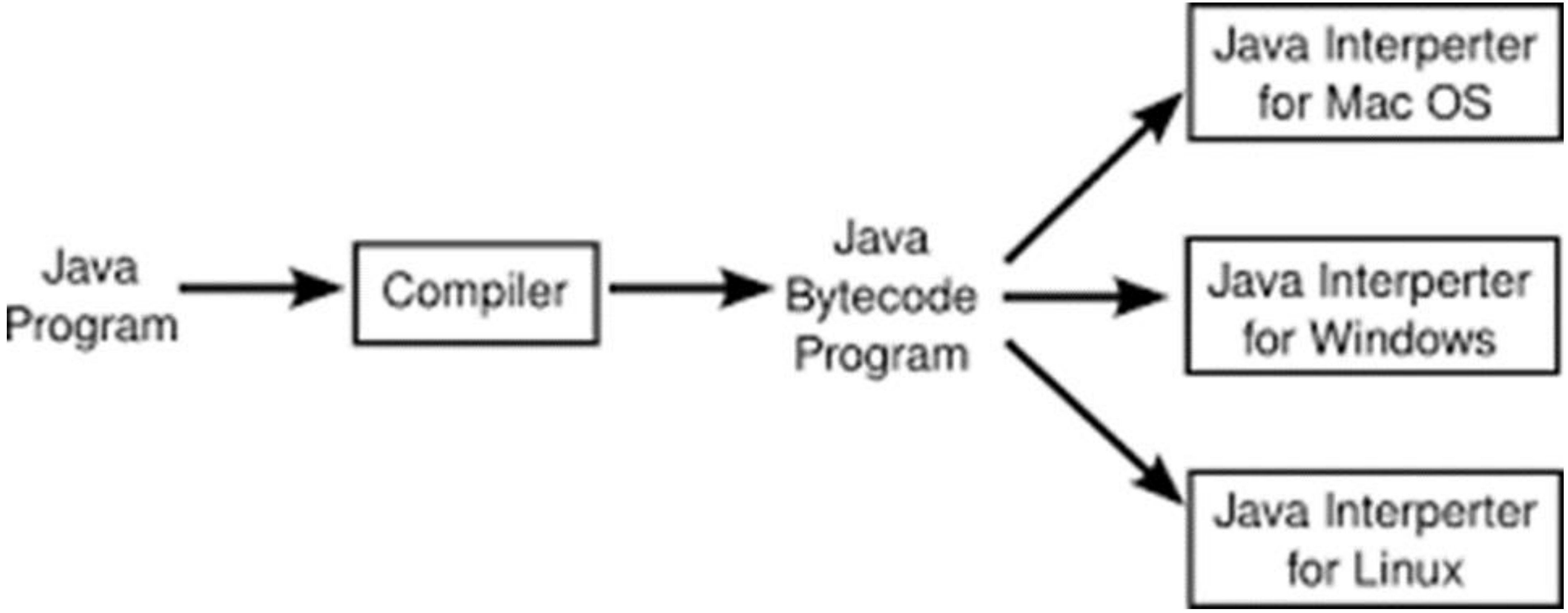


Virtual Machines

Step 1: Compilation from Java to Byte Code

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




Java Program
Compiler
Java Bytecode Program
Java Interperter for Mac OS
Java Interperter for Windows
Java Interperter for Linux

The Interpretation of Computer Programs

SECTION 4 COMPILATION STRATEGY

Compilation Strategies

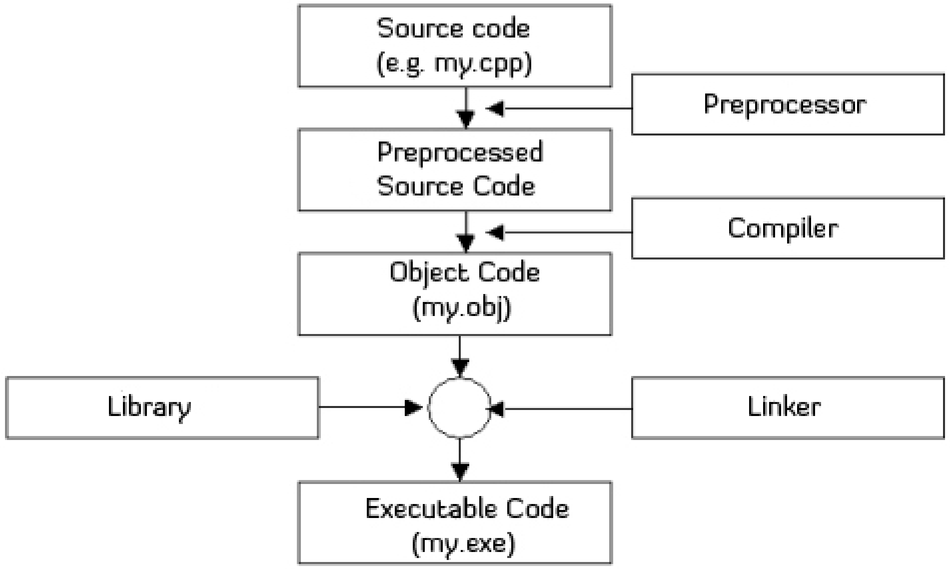
- Every language may use one or more compilation strategies.
- Examples are shown for certain language, but it may not be the only language to use that strategy.

(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)

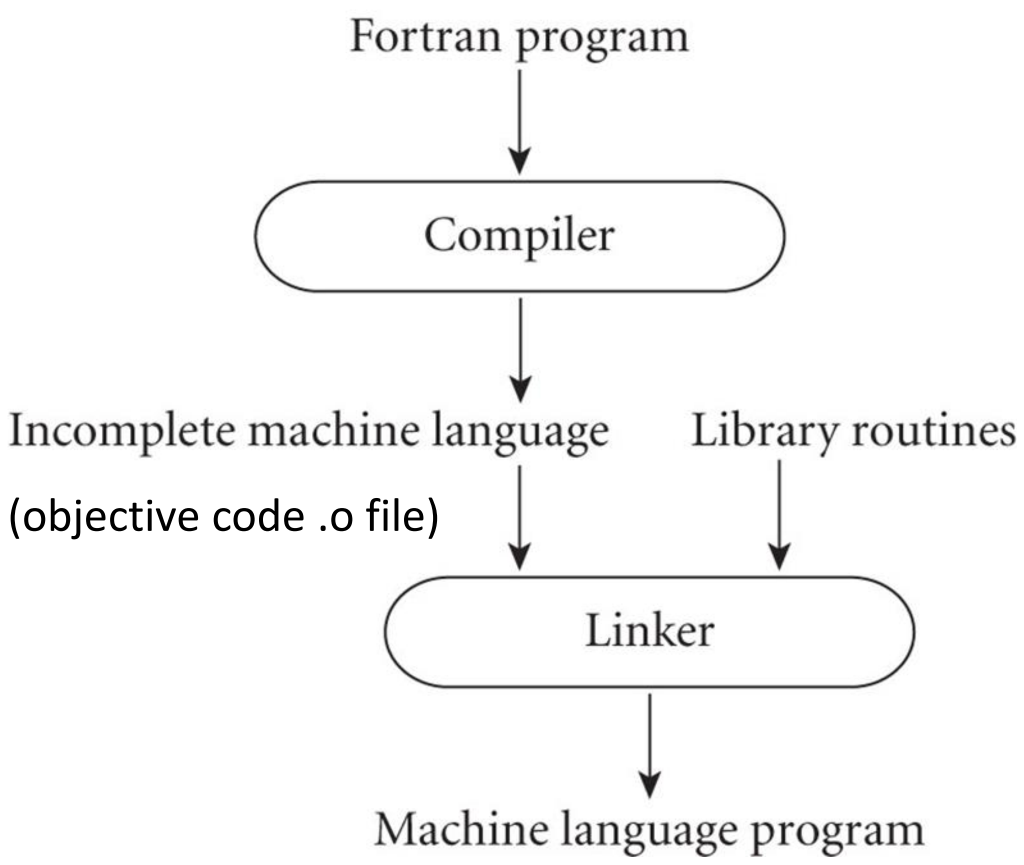
C/C++ Preprocessor



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

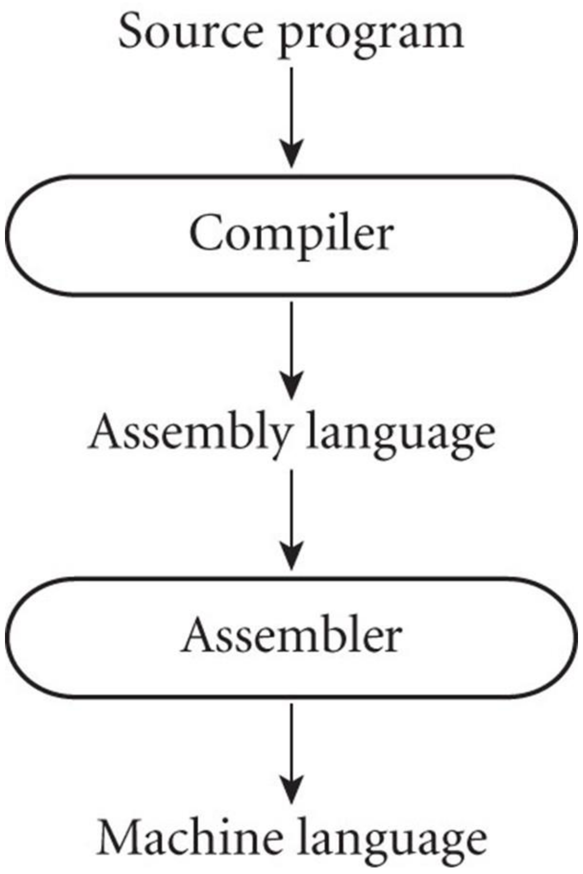
- Inclusion Directives **(#include)**
- Macro Definition Directives **(#define #undef)**
- Conditional Compilation Directives **(#if, #elif, #endif, #ifdef, #ifndef)**
- Other Directives **(#error, #line, #pragma)**

(B) Library of Routines and Linking



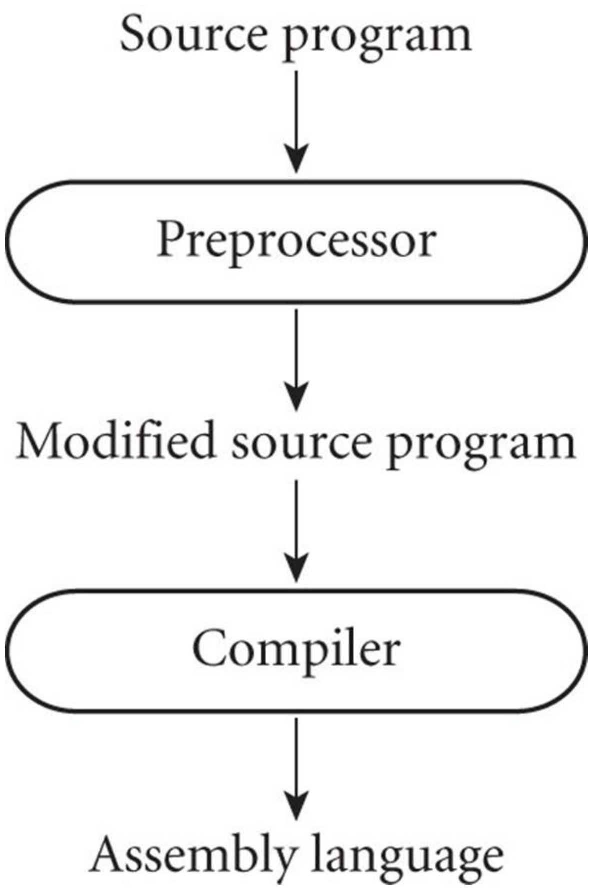
- Compiler uses a linker program to merge the appropriate library of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:

(C)Post-compilation Assembly



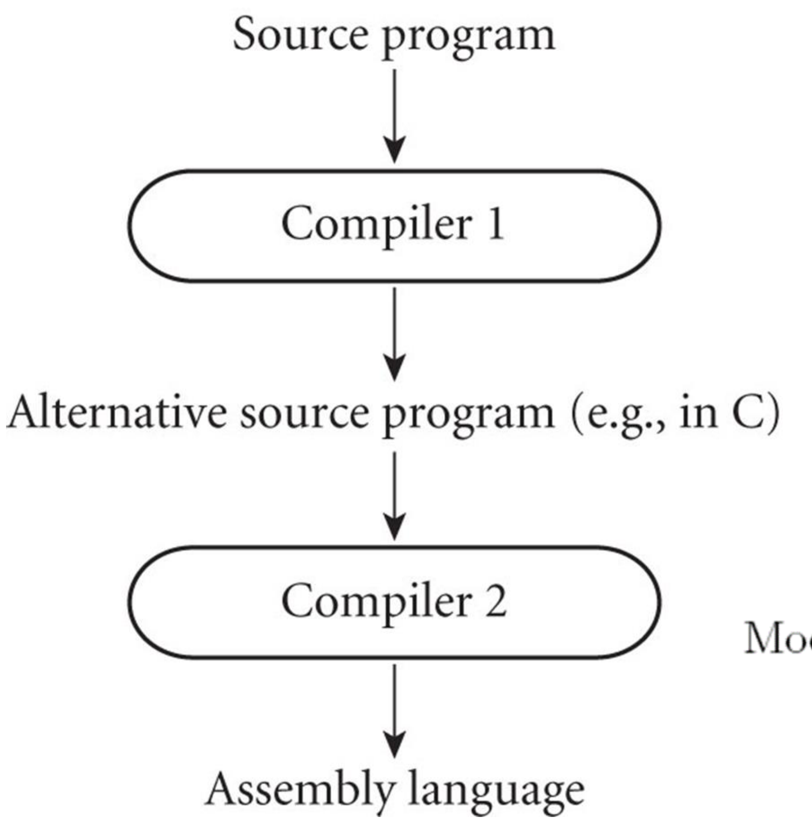
- Facilitates debugging (assembly language easier for people to read)
- Isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers)

(D) The C Preprocessor (conditional compilation)

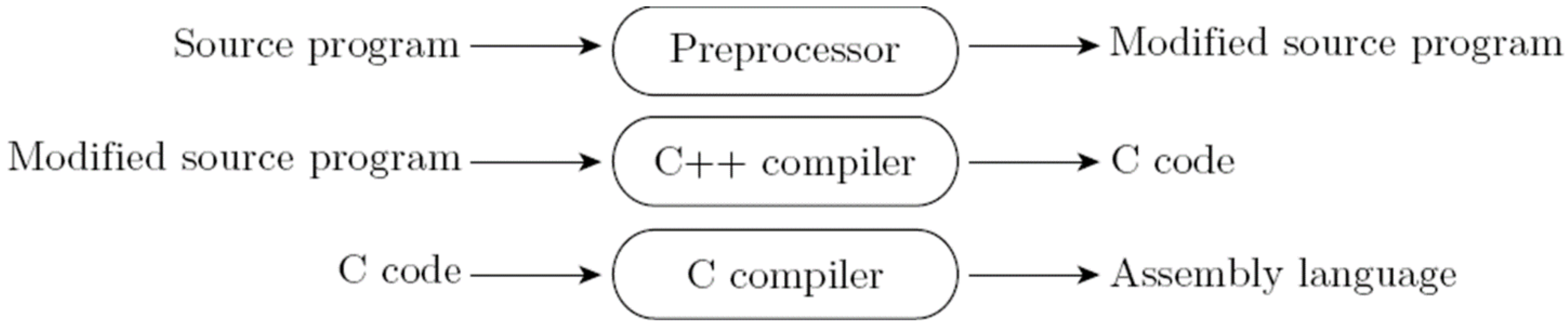


- Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source

(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)

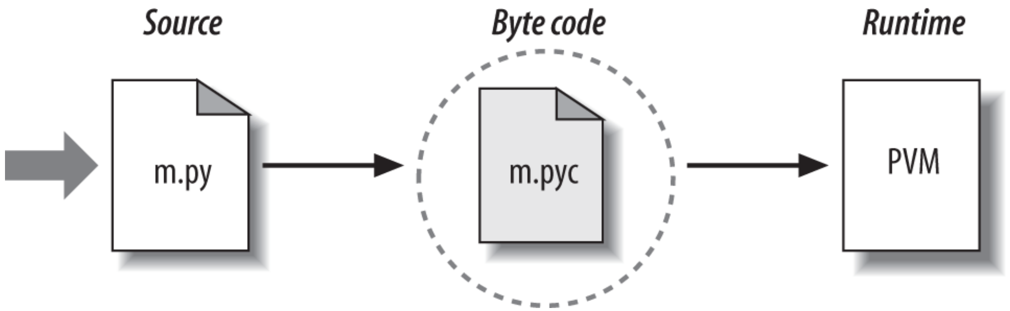


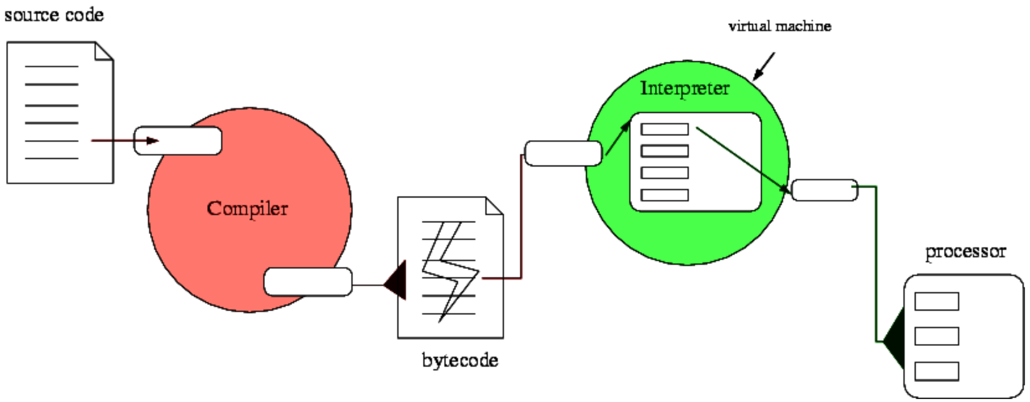
The Interpretation of Computer Programs

SECTION 5 INTRODUCTION STRATEGY

(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.

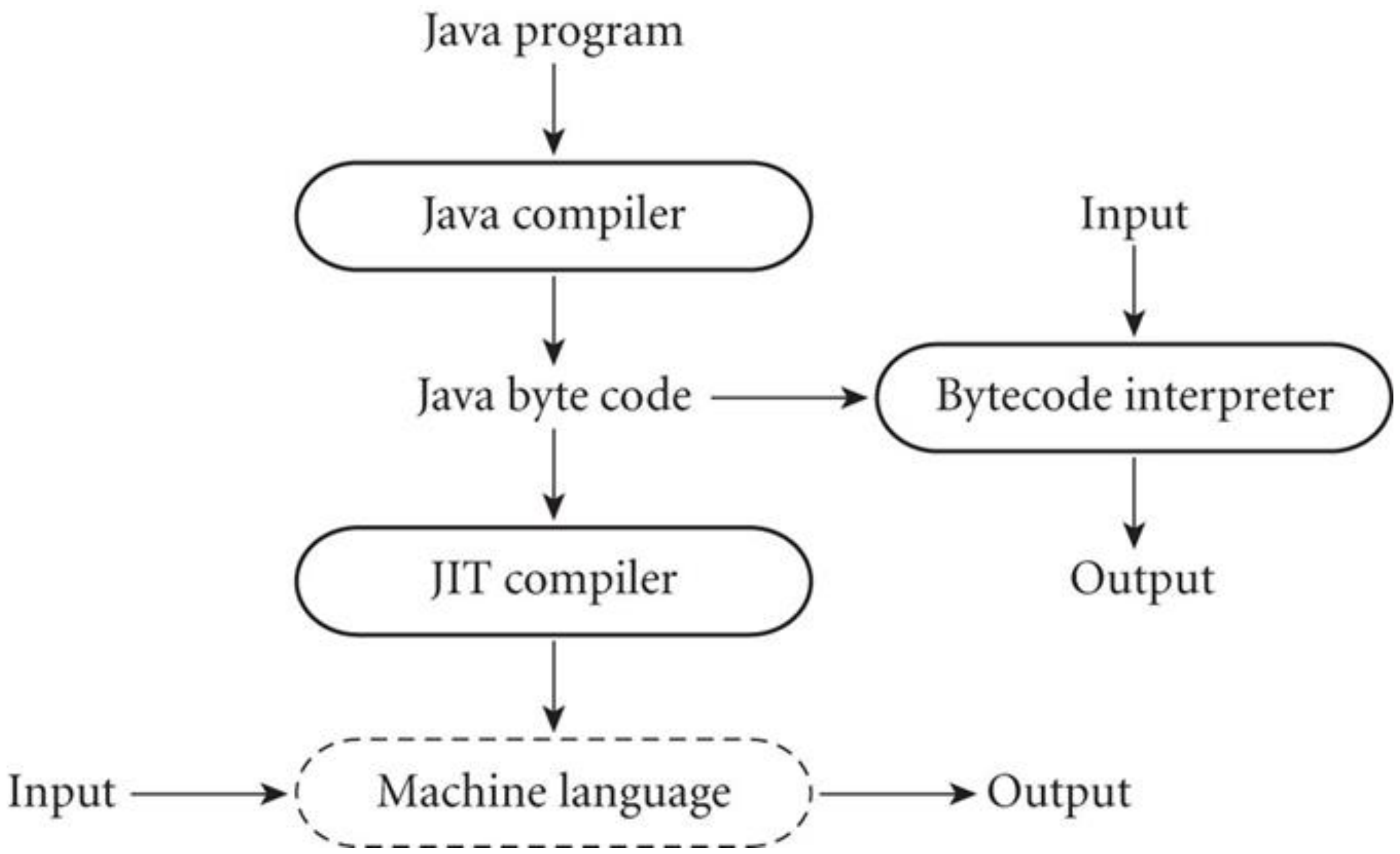


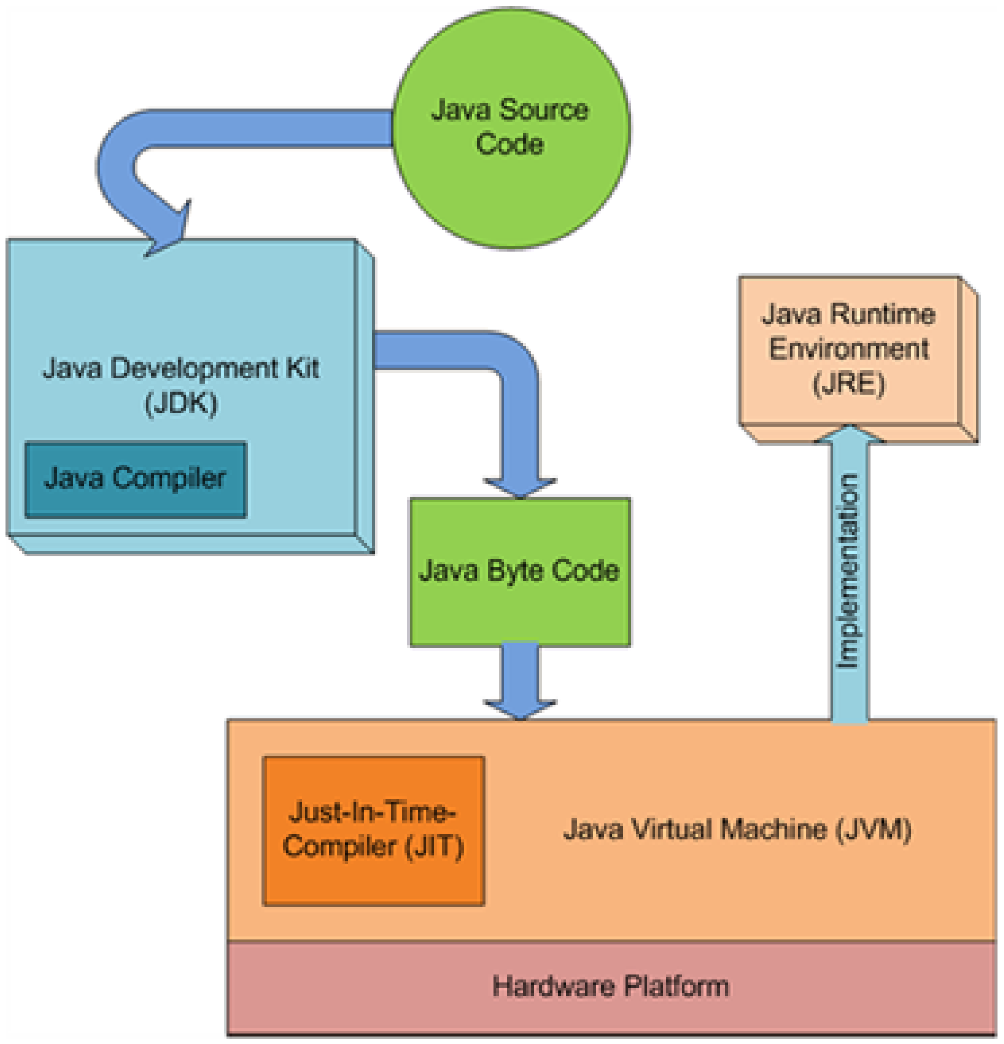


(G) Dynamic and Just-in-Time Compilation

- 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.

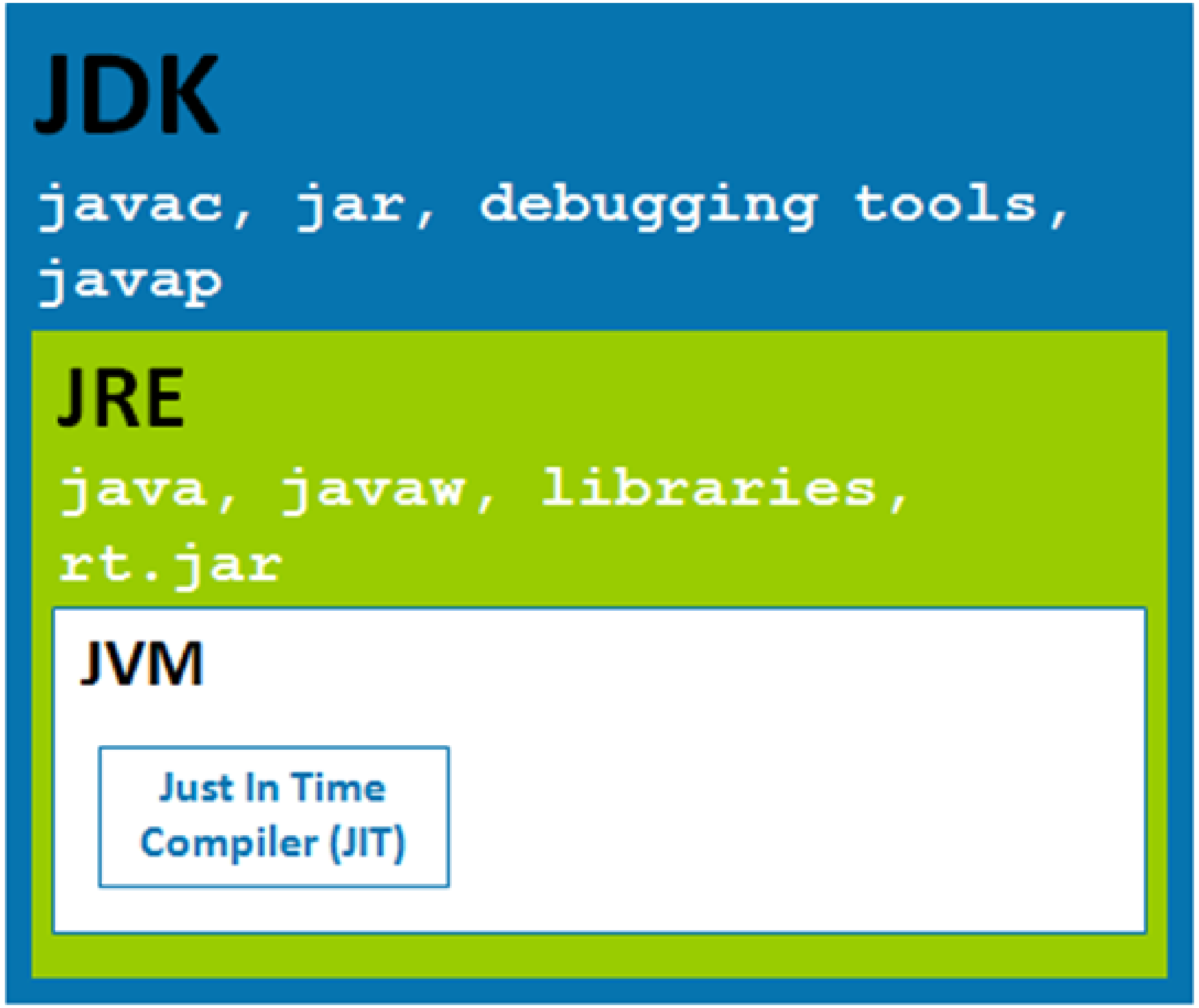
Java JIT


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



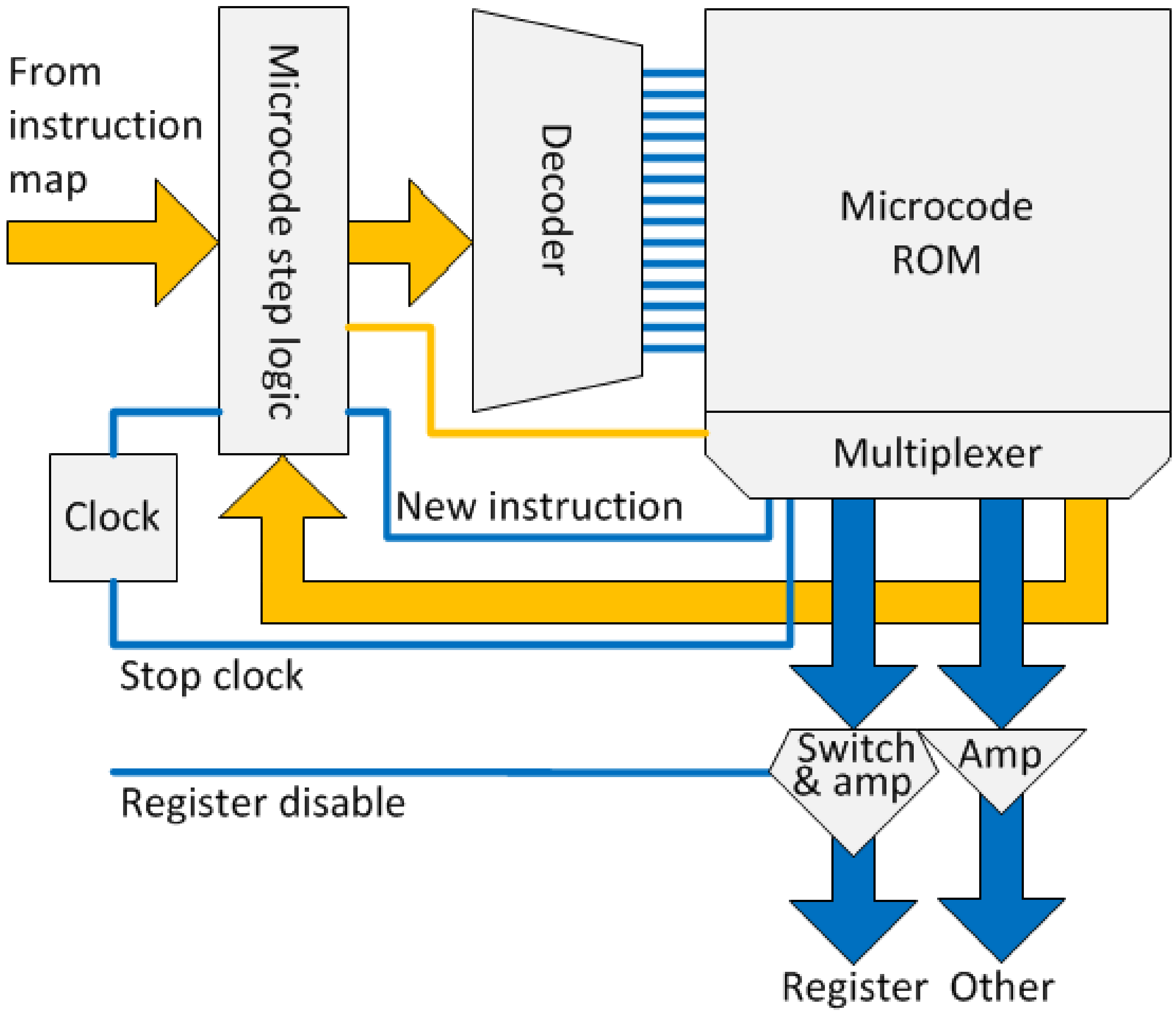
Just-In Time Compilation

Wait until you can no longer wait!


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.



Program Environment

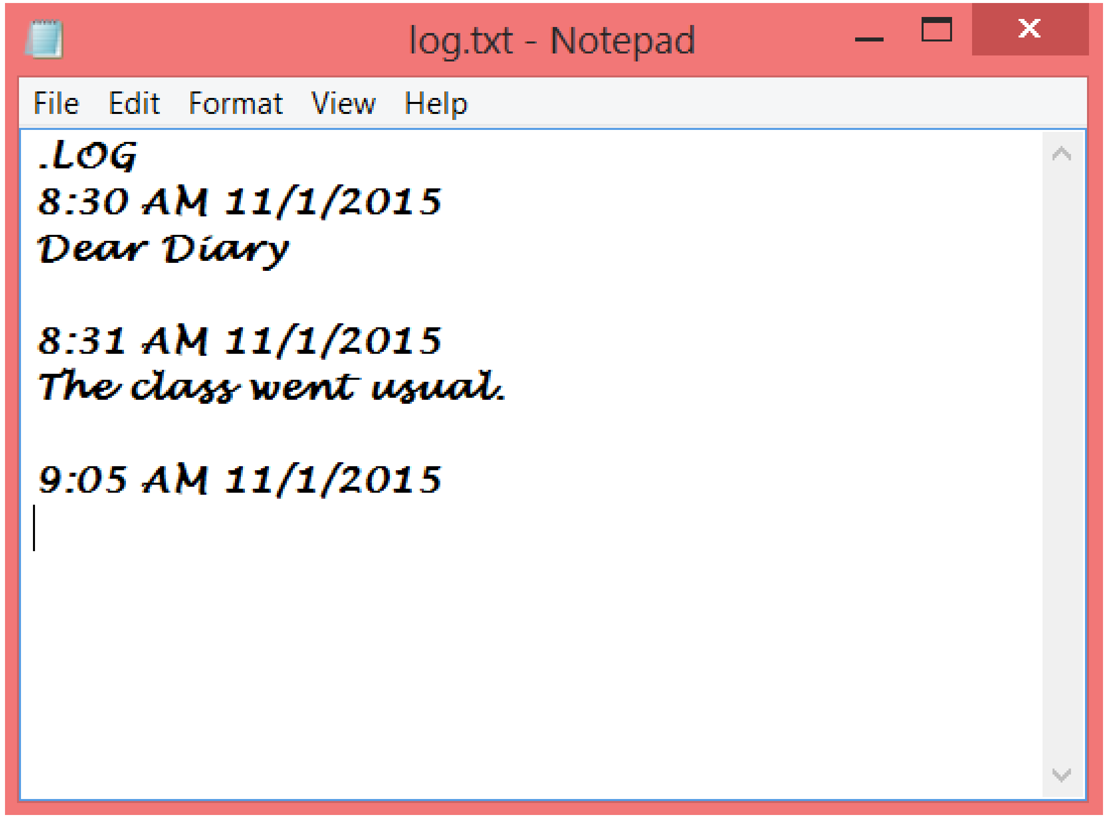
SECTION 6

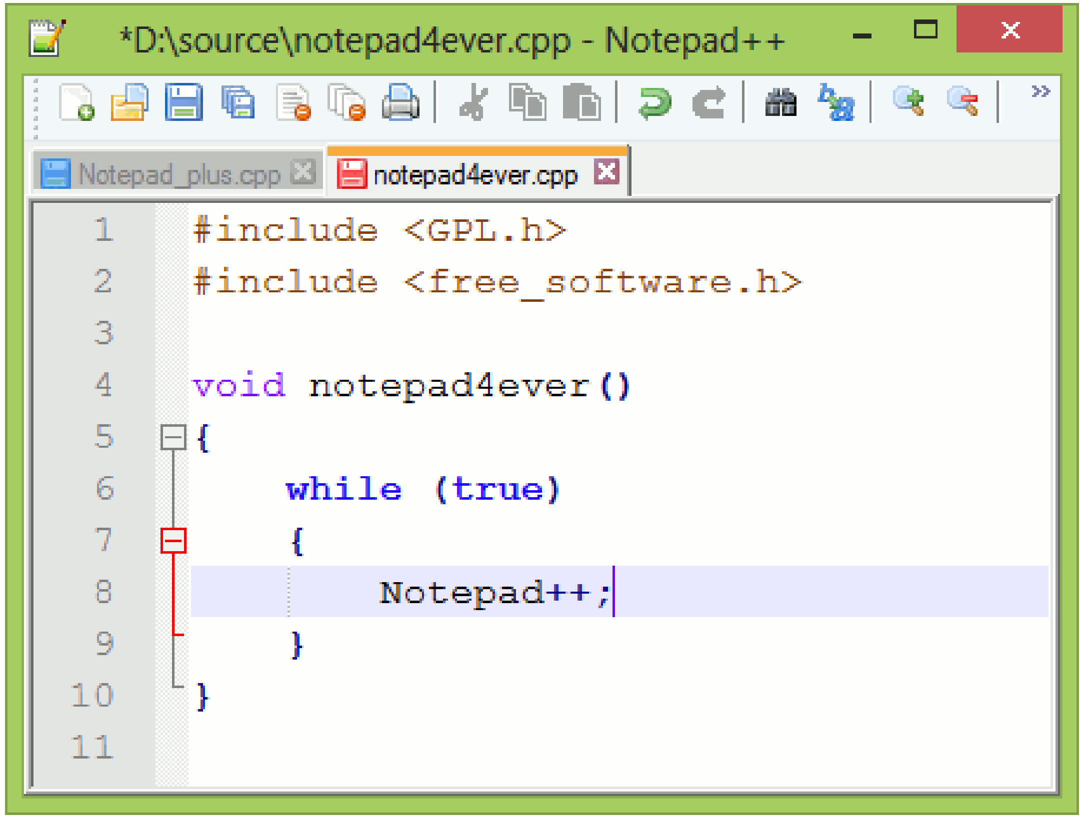
Unix Tools

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	Id, Id-so
Perusal tools	More, less, od, nm
Program cross-reference	ctags

Source Code Editor
Notepad, Notepad++

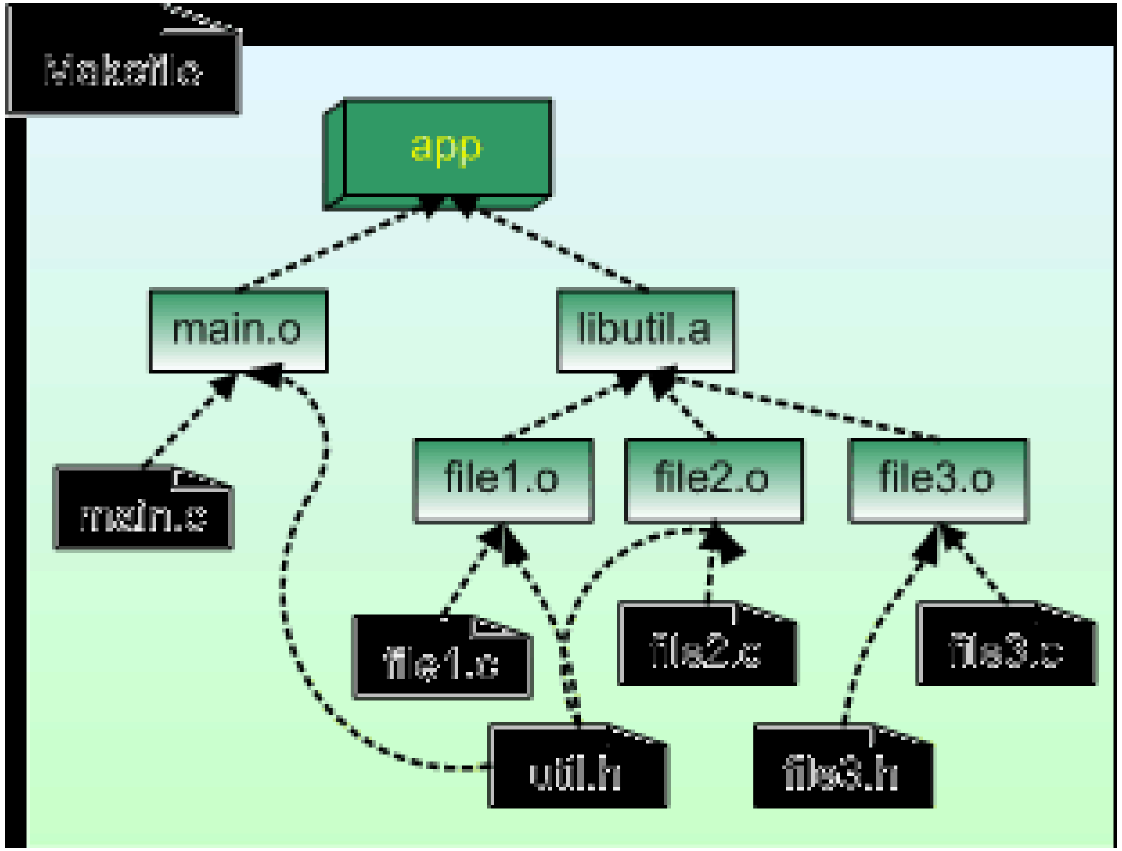




Integrated Development Environment

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

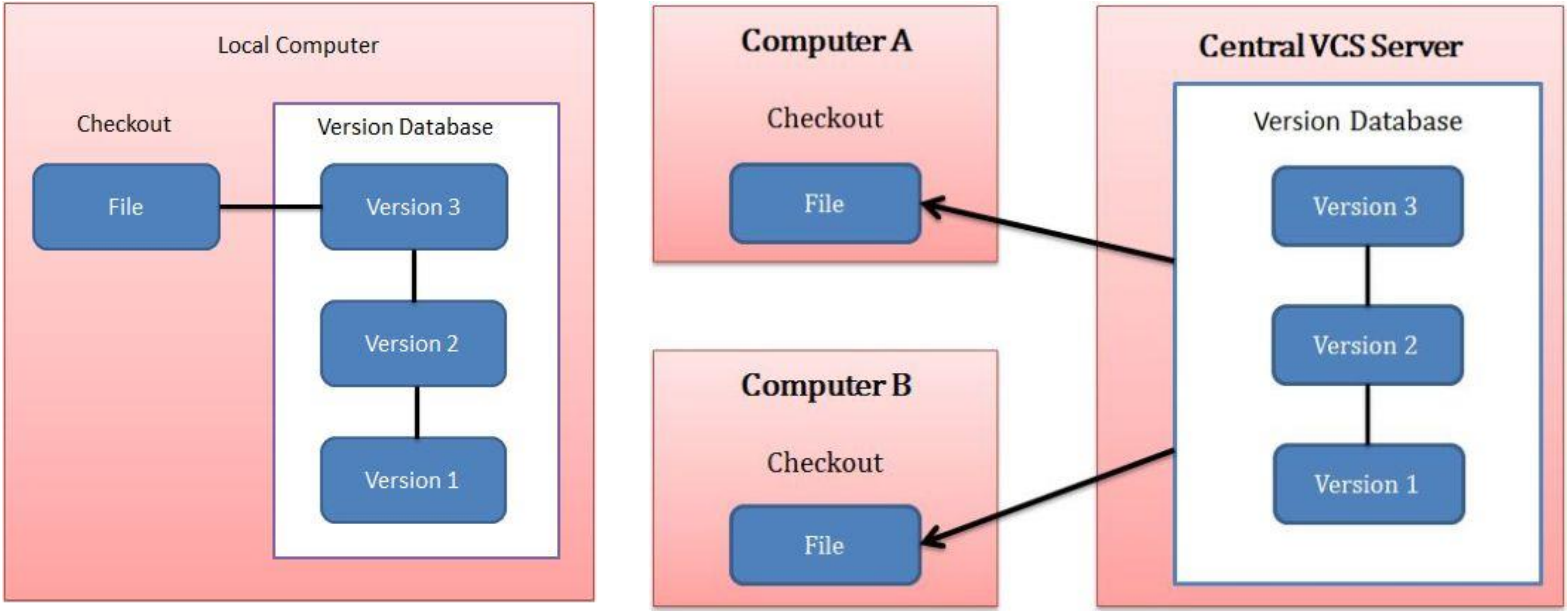
make: Program Builder



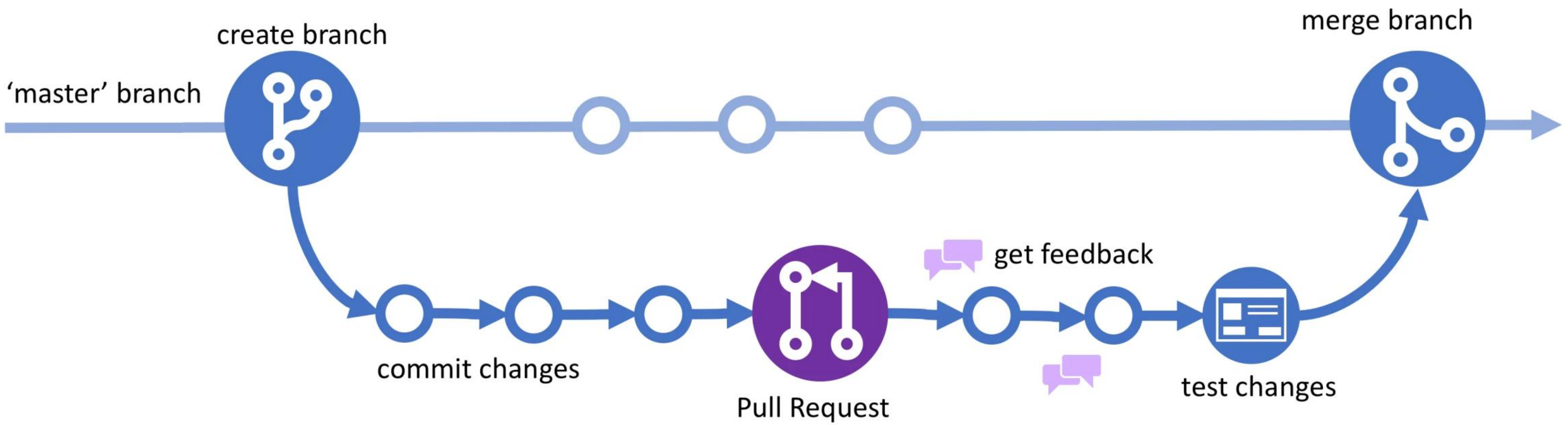

```
# define default target (first target = default)
# it depends on 'hello.o' (which will be created if necessary)
# and hello_func.o (same as hello.o)
hello: hello.o hello_func.o
        gcc -Wall hello.o hello_func.o -o hello
# define hello.o target
# it depends on hello.c (and is created from it)
# 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



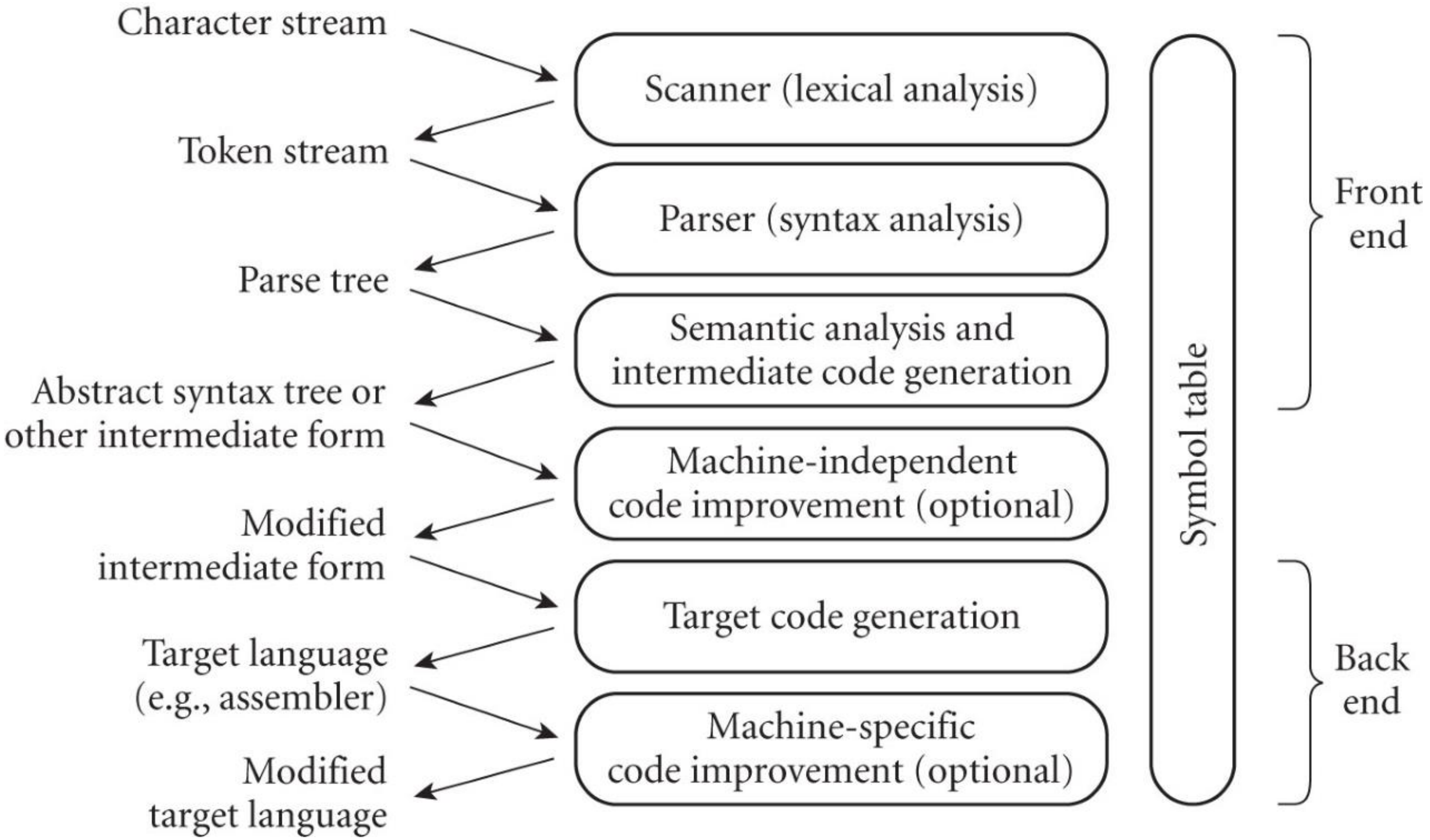
GitHub Flow


create branch
merge branch
'master' branch
get feedback
commit changes
Pull Request
test changes

Compilation: The Analysis of Computer Program Structures

SECTION 7 SIX PHASES

Phases of Compilation



Scanning

Lexical Analysis

- 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.

$M = (Q, \Sigma, \delta, q_0, F)$ where

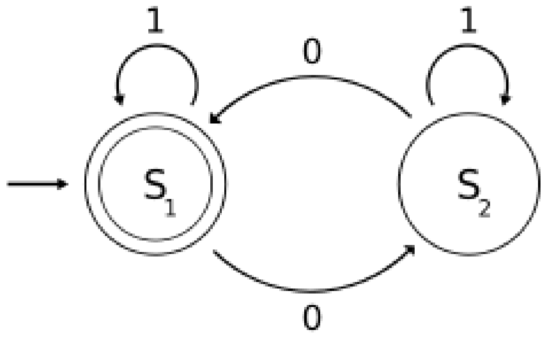
Input

State

State Transition

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

	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**

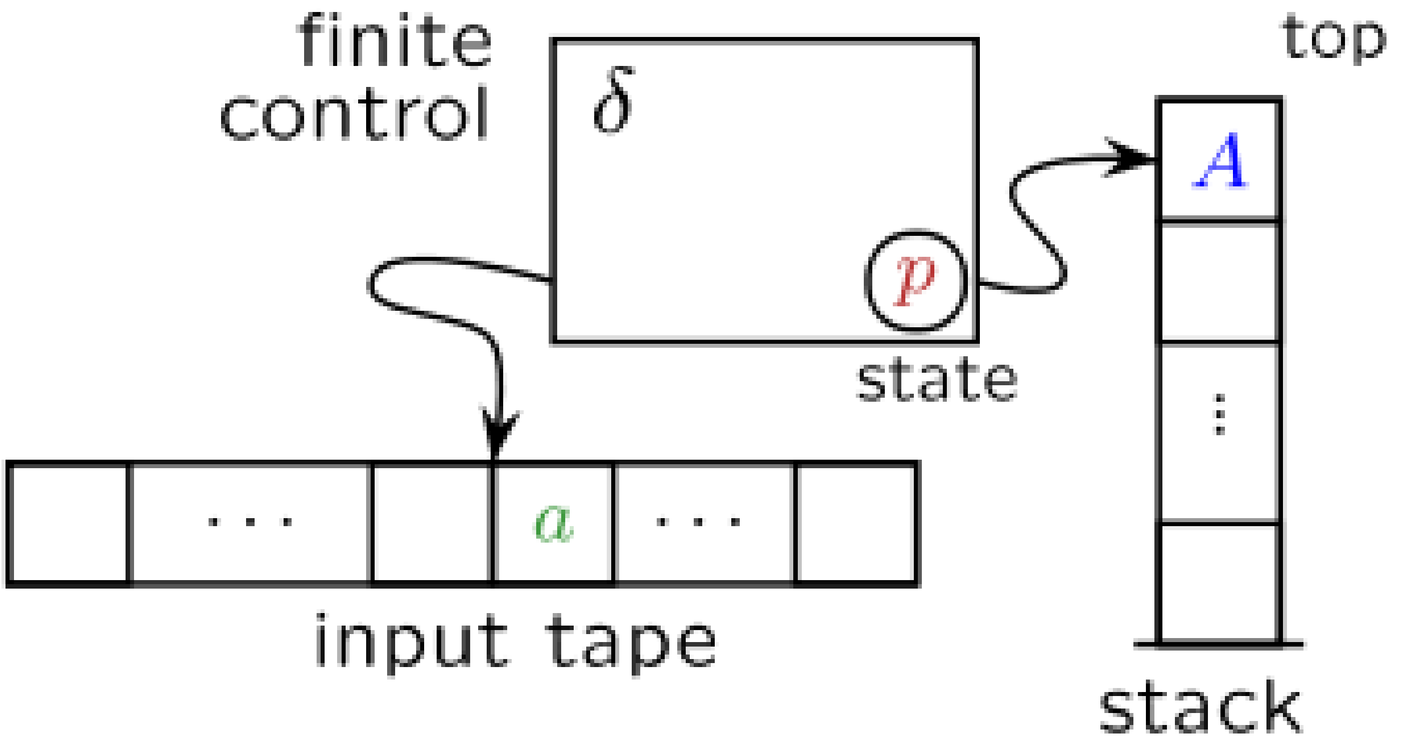
- Parsing discovers the "context free" structure of the program
- Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)

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

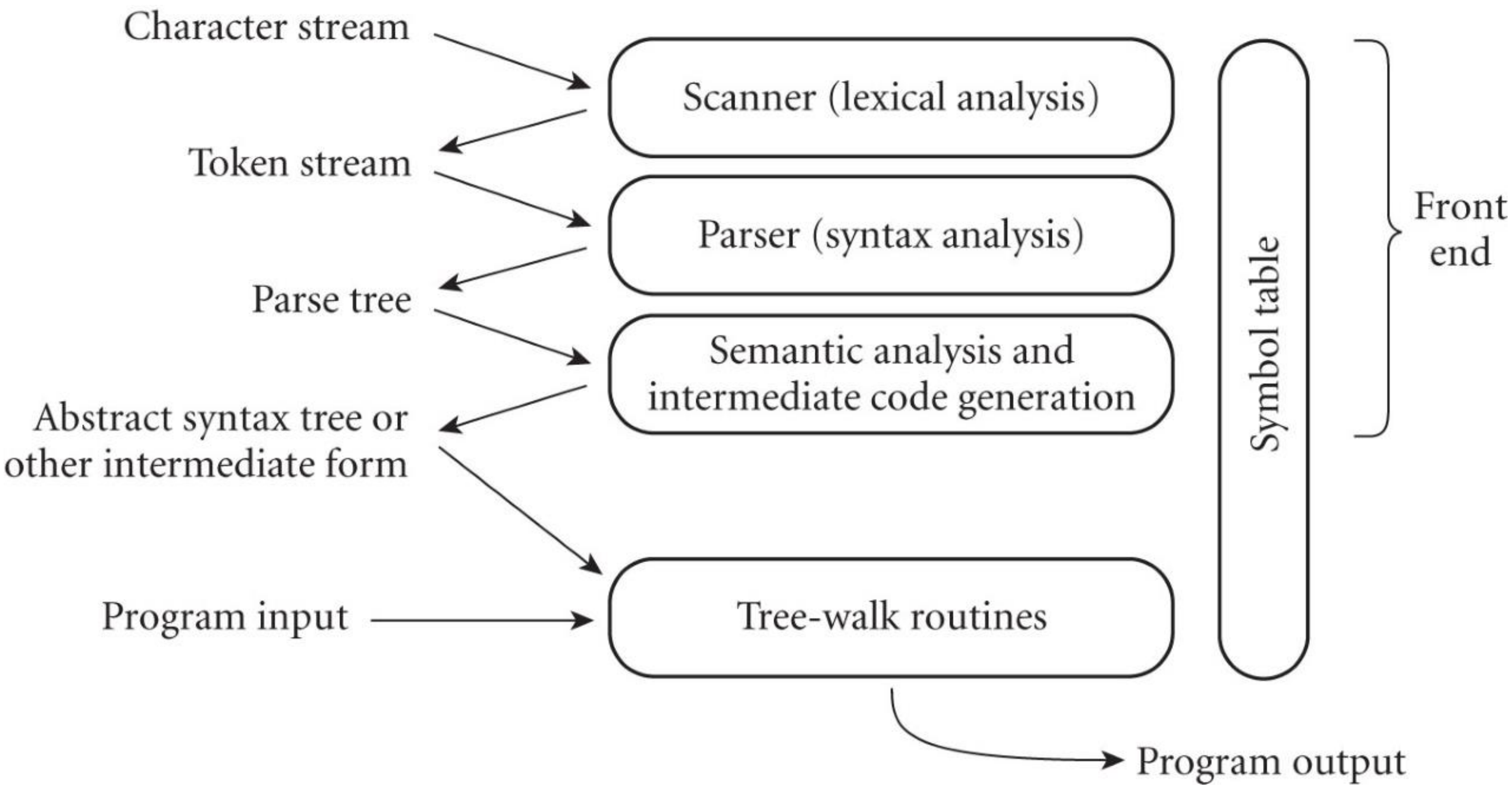
Code Generation

Symbol Table

Compilation: The Analysis of Computer Program Structures

SECTION 8 FRONT-END COMPILER

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

Context-Free Grammar and Parsing

Example (while loop in C)

$iteration\text{-}statement \longrightarrow while\ (\ expression\ )\ statement$

statement, in turn, is often a list enclosed in braces:

$statement \longrightarrow compound\text{-}statement$

$compound\text{-}statement \longrightarrow \{\ block\text{-}item\text{-}list\ opt\ \}$

where

$block\text{-}item\text{-}list\ opt \longrightarrow block\text{-}item\text{-}list$

or

$block\text{-}item\text{-}list\ opt \longrightarrow \epsilon$

and

$block\text{-}item\text{-}list \longrightarrow block\text{-}item$

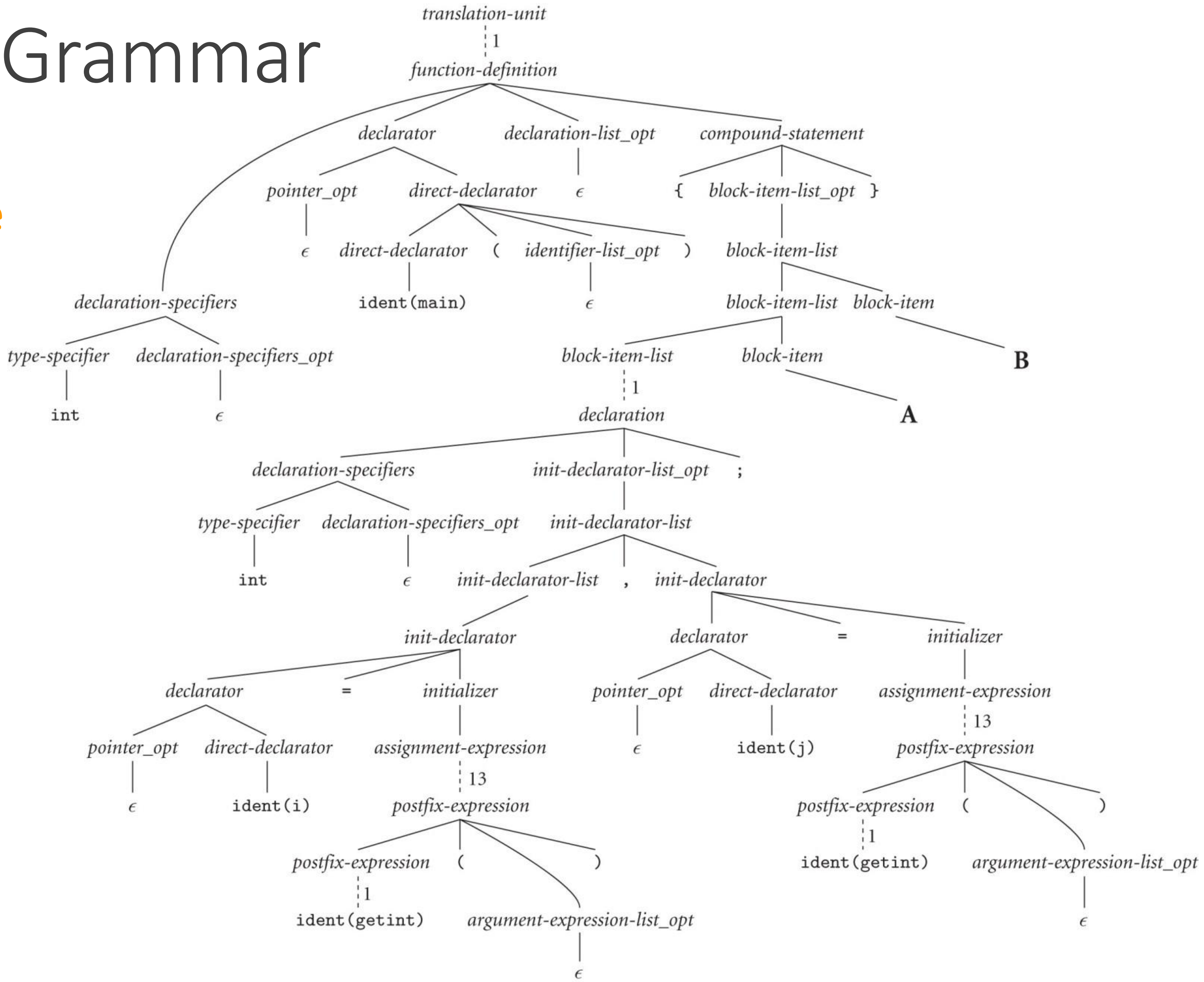
$block\text{-}item\text{-}list \longrightarrow block\text{-}item\text{-}list\ block\text{-}item$

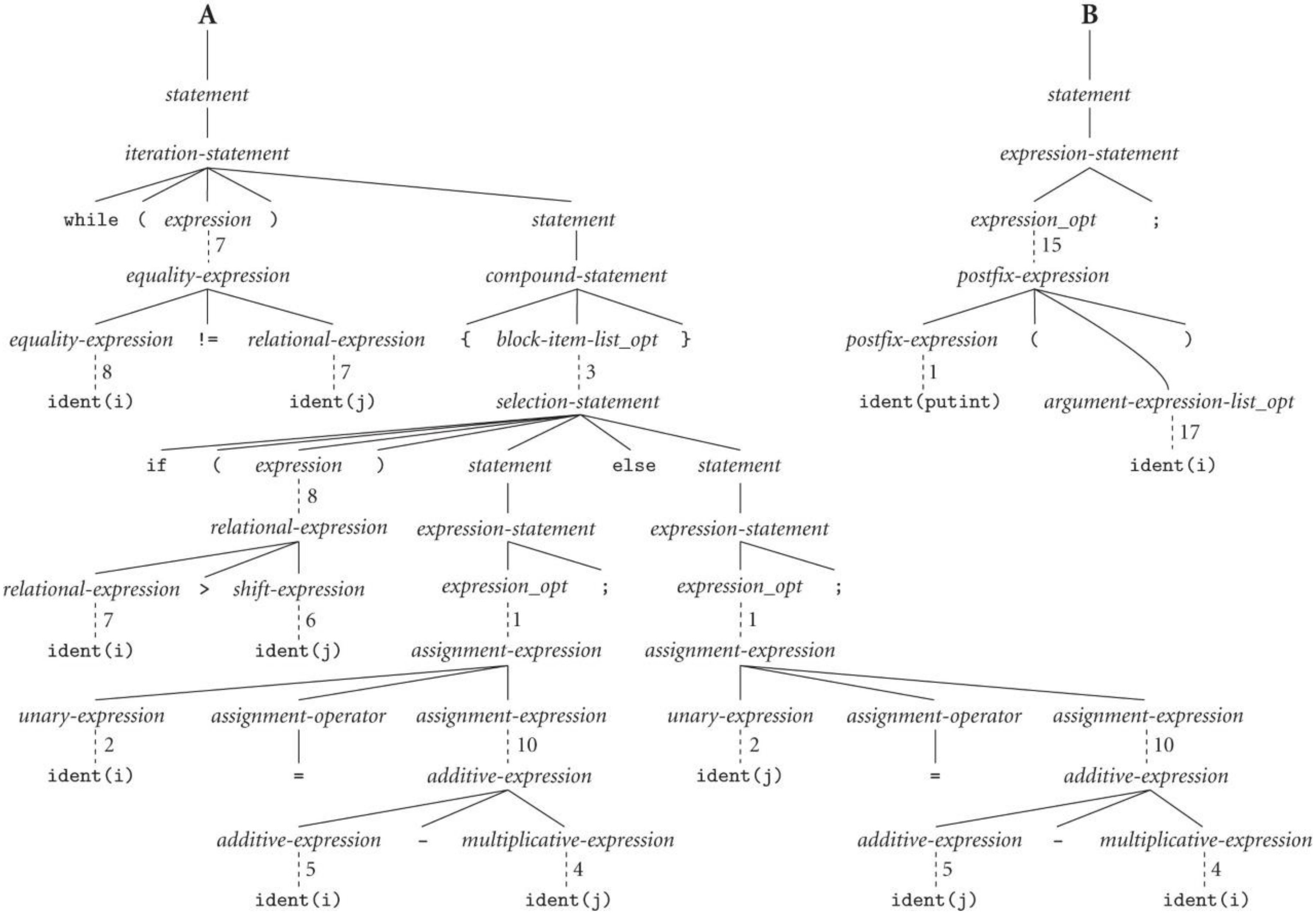
$block\text{-}item \longrightarrow declaration$

$block\text{-}item \longrightarrow statement$

Context-Free Grammar and Parsing

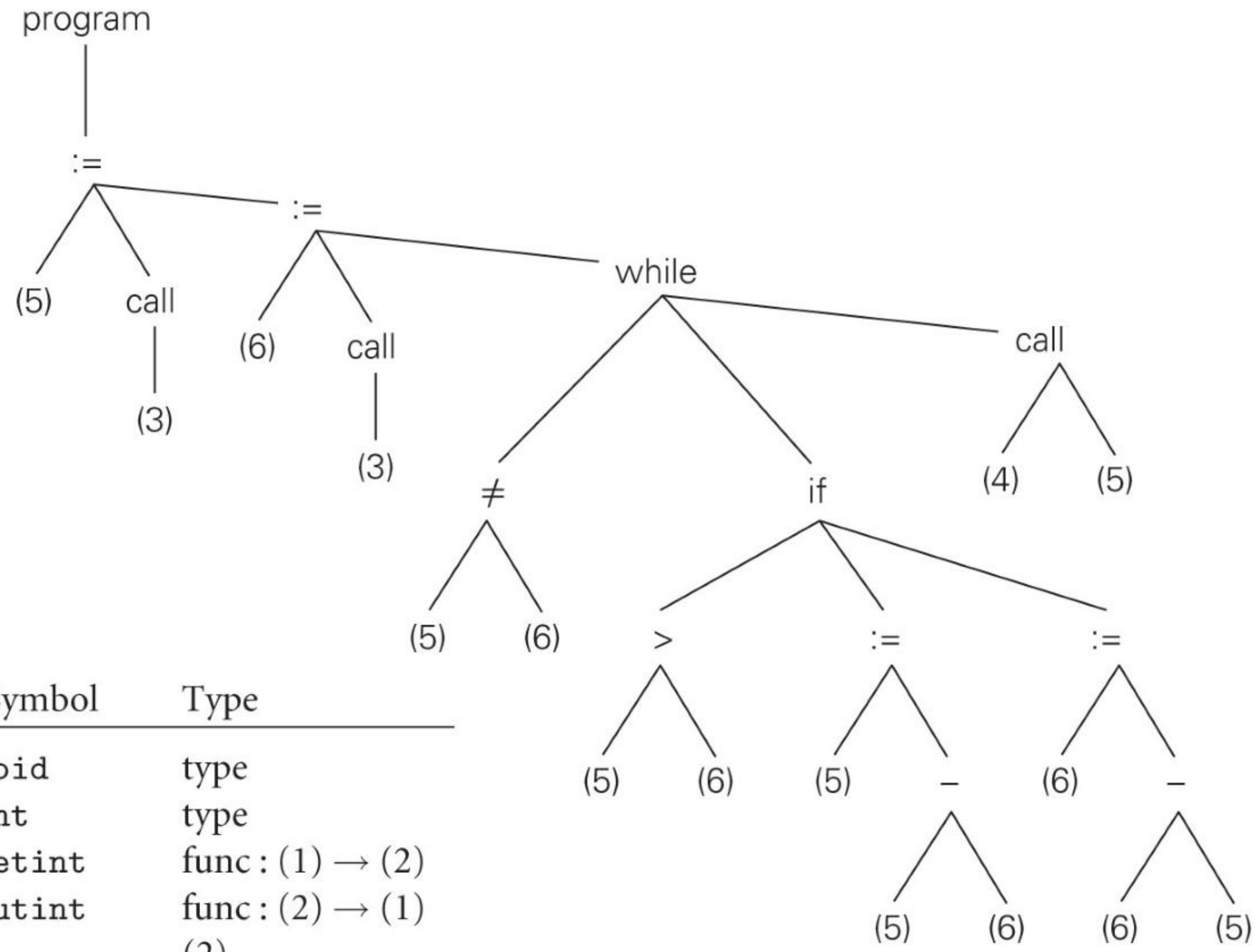
GCD Program Parse Tree




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

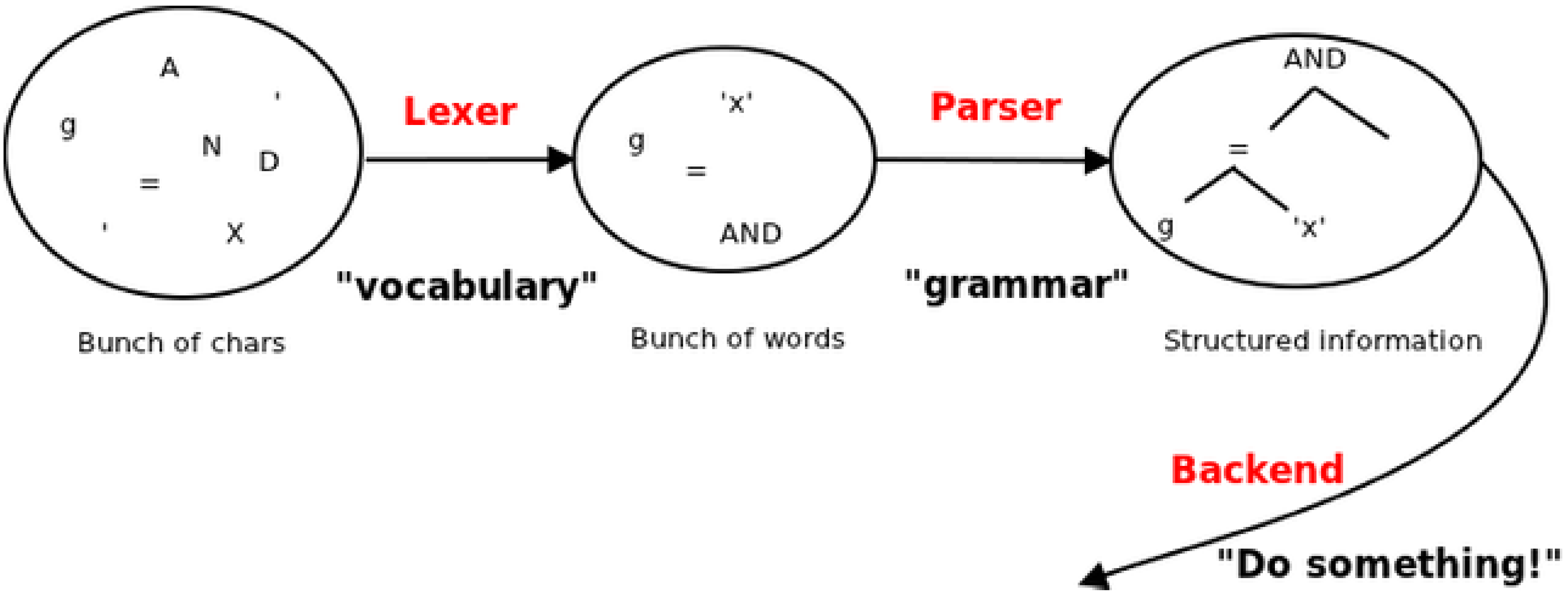
GCD Program Parse Tree

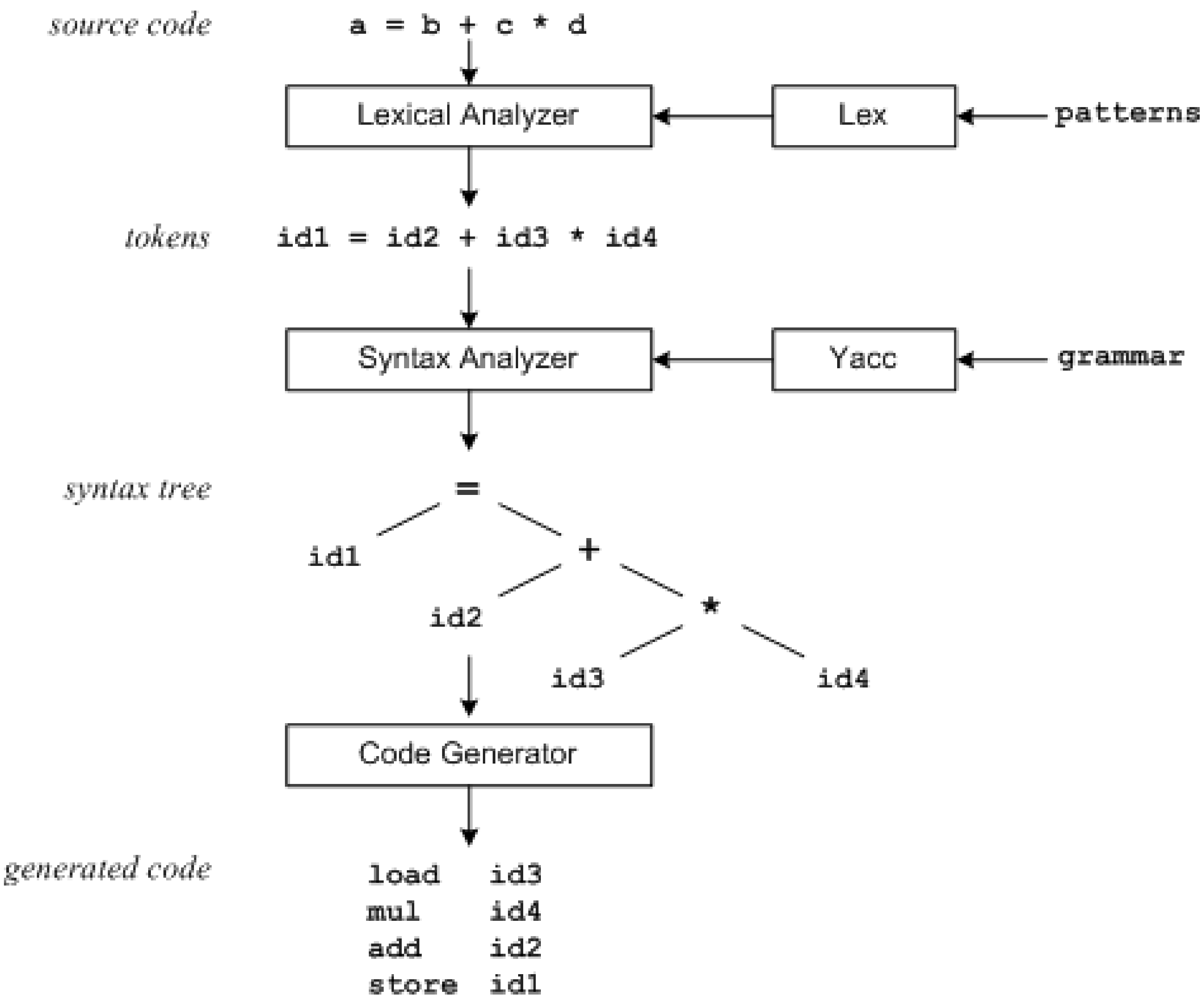


Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

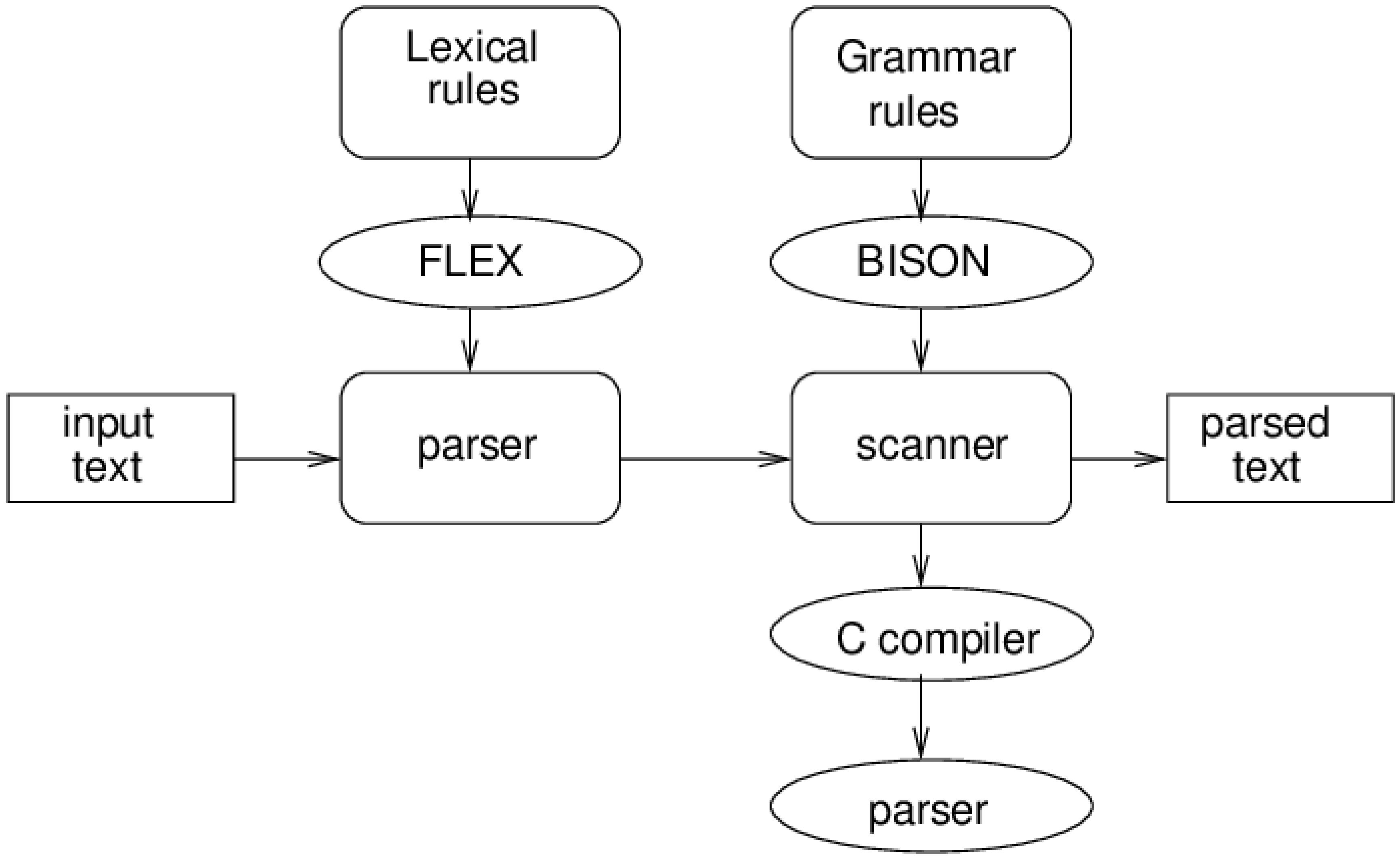
The Design of Compiler Or Interpreter

SECTION 9

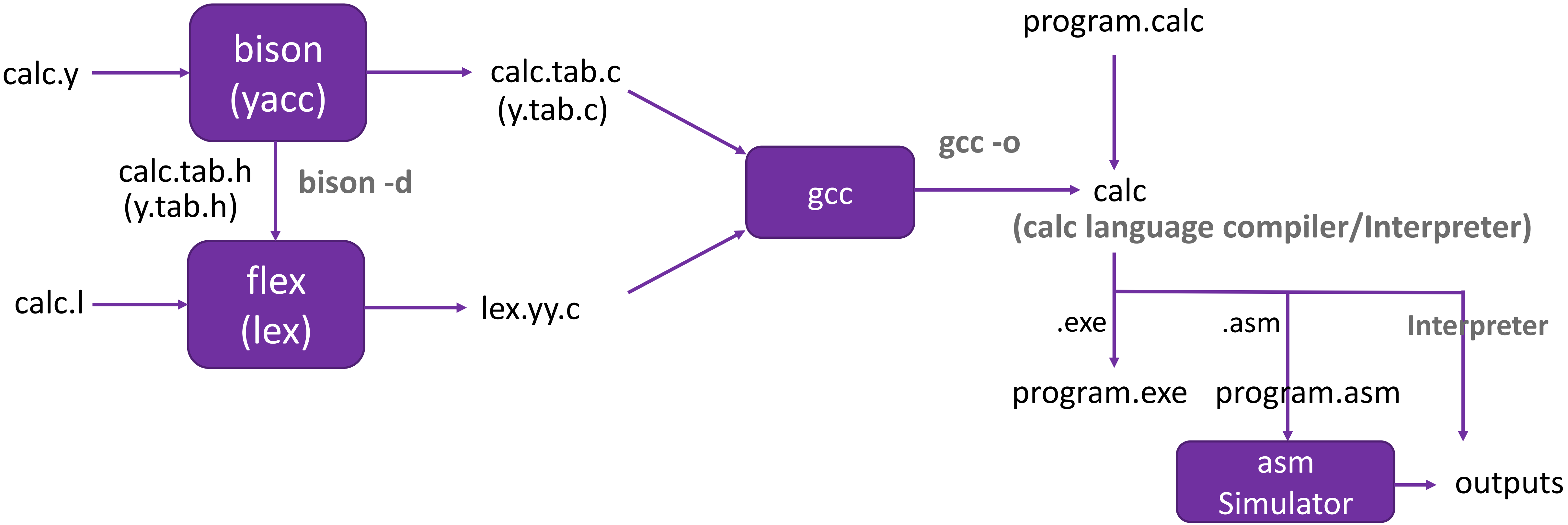

A
'
g
N
D
=
'
X
Lexer
"vocabulary"
Bunch of chars
'x'
g
=
AND
Bunch of words
Parser
"grammar"
AND
=
g
'x'
Structured information
Backend
"Do something!"

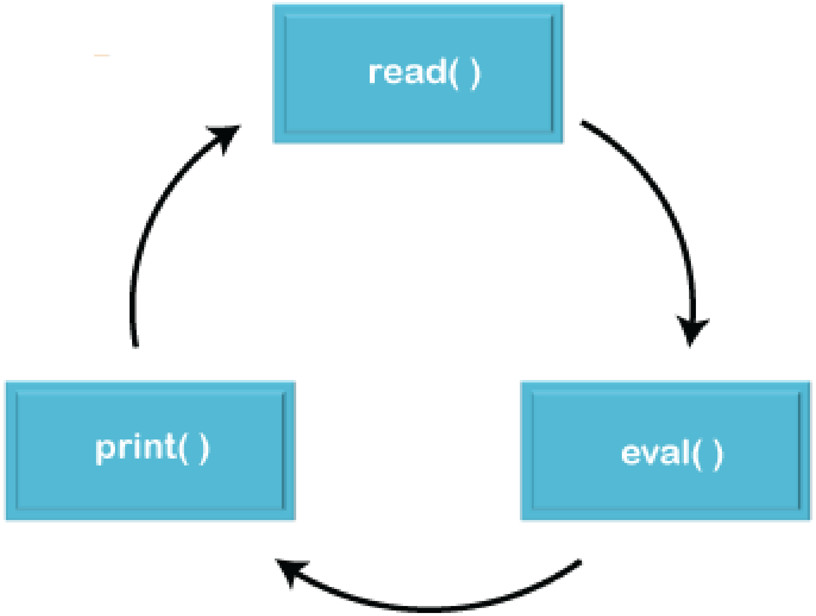


source code
a = b + c * d
Lexical Analyzer
Lex
patterns
tokens
id1 = id2 + id3 * id4
Syntax Analyzer
Yacc
grammar
syntax tree
=
id1
+
id2
*
id3
id4
Code Generator
generated code
load id3
mul id4
add id2
store id1

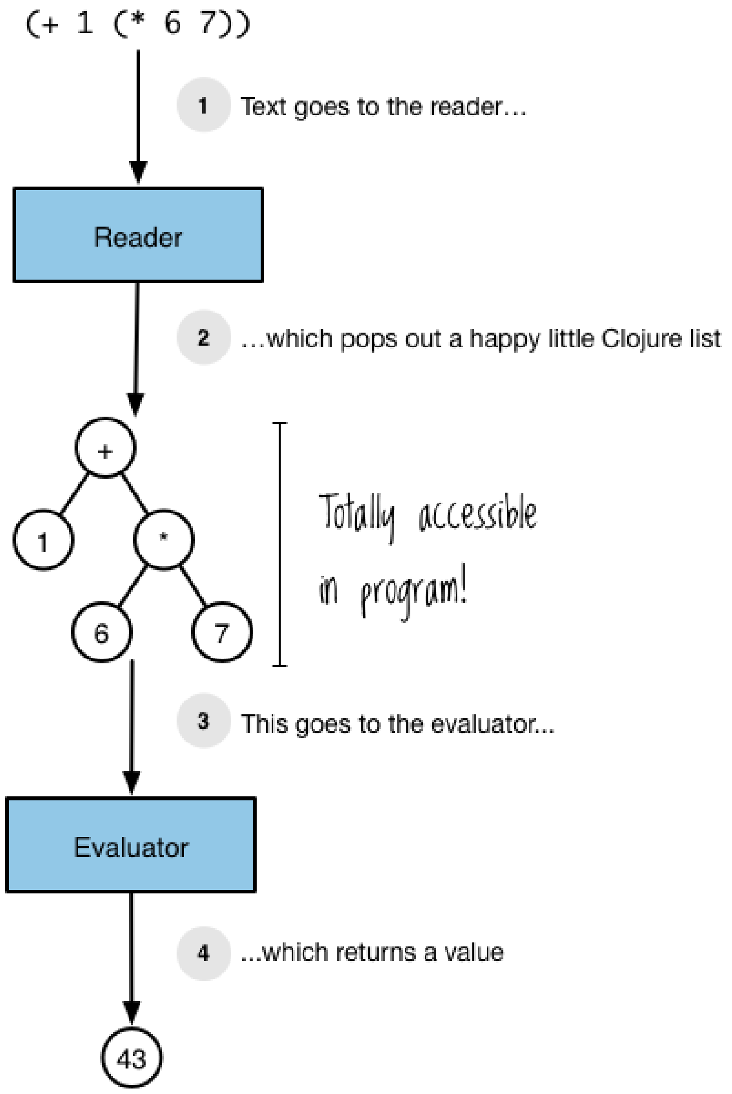

Lexical rules
Grammar rules
FLEX
BISON
input text
parser
scanner
parsed text
C compiler
parser

How Flex and Bison Works Together?





Read-Eval-Print Loop (REPL)



Key REPL Parts

Program Structure Representation (PSR)

Parsing: From input the PSR

Tokenization: Serializing PST as text that can be read by READ

Environment: Representation plus defining, setting and looking up variables

Function representation

Evaluation of PSR

REPL (read, eval, print) Loop

End of Chapter 1