Programming Language and Paradigms

Introduction

Tasanawan Soonklang
Department of Computing, Silpakorn University

Chapter 1 Topics

- A brief history of programming languages
- Why study programming languages?
- The art of language design
- Language evaluation criteria
- Programming language paradigms
- Implementation methods
- Programming environments

A Brief History of Programming Languages

Brief history: Machine language

- Machine language – the sequence of bits that directly controls a processor
- Add, compare, move data from one place to another, and so forth at appropriate times

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31 00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6 00 00 00 00 39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

GCD program in machine language for the x86 (Pentium) instruction set, expressed as hexadecimal (base 16) numbers

Brief history: Assembly language

- Assembly language – expressed with mnemonic abbreviations, a less error-prone notation

```
        pushl   % ebp                   jle     D
        movl    %esp, %ebp              subl    %eax, %ebx
        pushl   %ebx            B:      cmpl    %eax, %ebx
        subl    $4, %esp                jne         A
        andl    $-16, %esp      C:      movl    %ebx, (%esp)
        call    getint                  call    put int
        movl    %eax, %ebx              movl    -4(%ebp), %ebx
        call    getint                  leave
        cmpl    %eax, %ebx              ret
        je      C               D:      subl    %ebx, %eax
A:      cmpl    %eax, %ebx              jmp     B
```

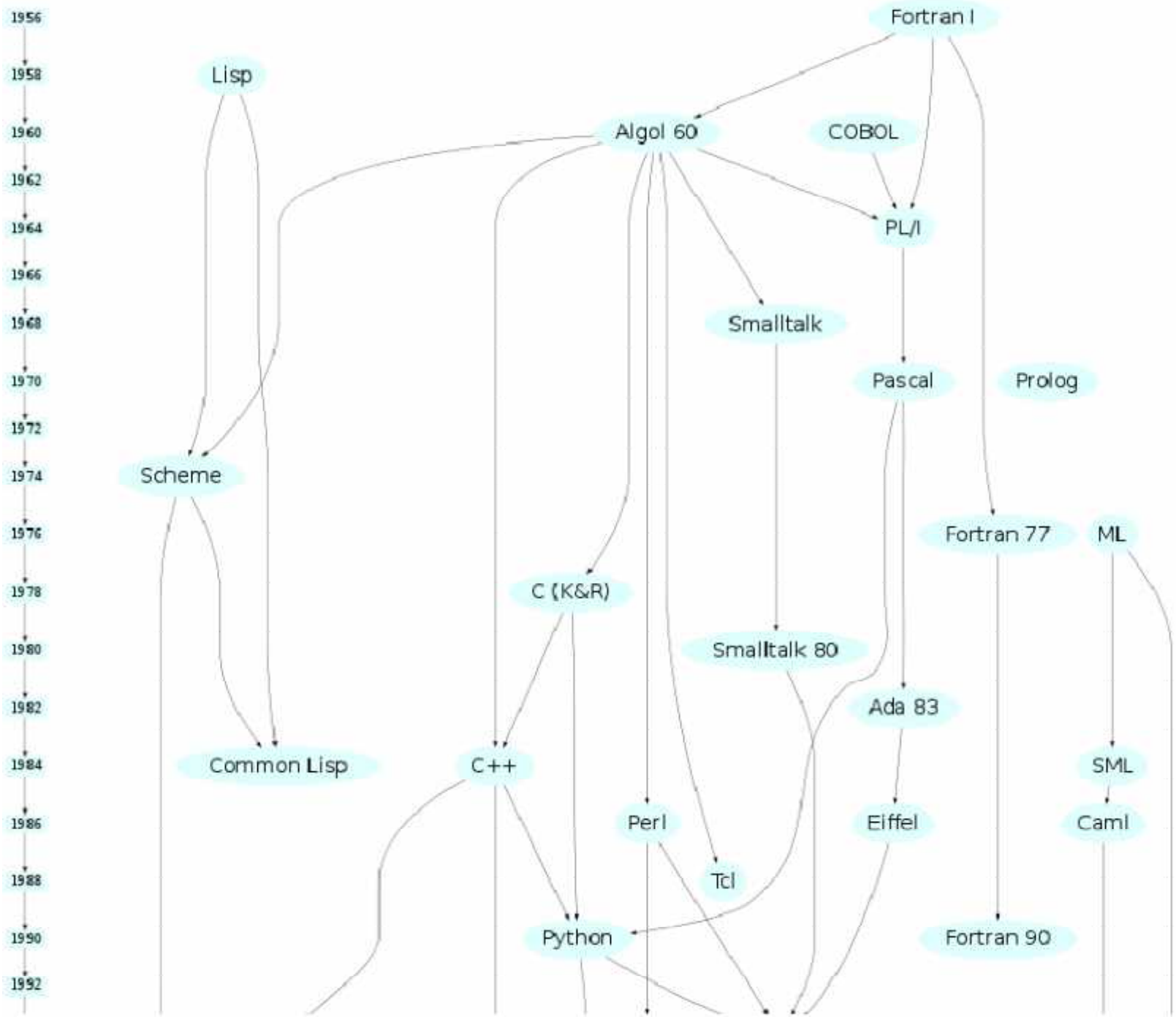
GCD program in assembly language for the x86.

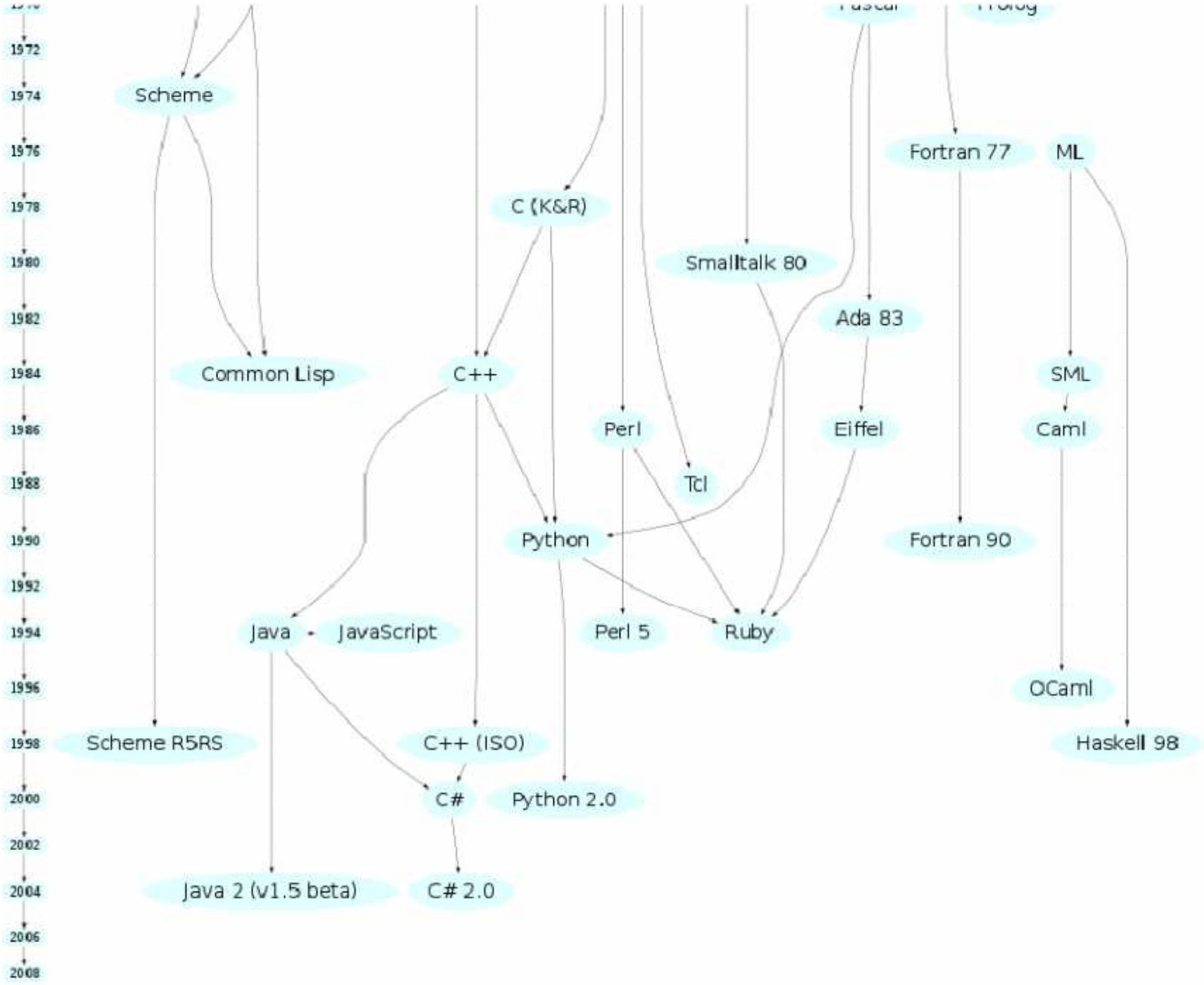
Brief history: Assembly language

- One-to-one correspondence between mnemonics and machine language instructions
- Assembler – system program for translating from mnemonics to machine language
- Machine-dependent language – rewrite programs for every new machine
- Difficult to read and write large programs

Brief history: high-level language

- Fortran – first high-level language in the mid-1950s
- Machine-independent language
- Compiler – system program for translating from high-level language to assembly or machine language
- Not one-to-one correspondence between source and target operations.


1956
1958
1960
1962
1964
1966
1968
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
Fortran I
Lisp
Algol 60
COBOL
PL/I
Smalltalk
Pascal
Prolog
Scheme
Fortran 77
ML
C (K&R)
Smalltalk 80
Ada 83
Common Lisp
C++
SML
Perl
Eiffel
Caml
Tcl
Python
Fortran 90


1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
Scheme
Fortran 77
ML
C (K&R)
Smalltalk 80
Ada 83
Common Lisp
C++
SML
Perl
Eiffel
Caml
Tcl
Python
Fortran 90
Java
JavaScript
Perl 5
Ruby
OCaml
Scheme R5RS
C++ (ISO)
Haskell 98
C#
Python 2.0
Java 2 (v1.5 beta)
C# 2.0

Why Study Programming Languages?

Why study programming languages?

- Understand obscure features
- Choose among alternative ways to express things
- Simulate useful features in languages that lack them
- Make it easier to learn new languages
- Help you choose a language

Why study programming languages?

- Understand obscure features
  - In C, help you understand unions, arrays & pointers, separate compilation, catch and throw
  - In C++, help you understand multiple inheritance, * operator
  - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals

Why study programming languages?

- Choose among alternative ways to express things
  - understand implementation costs:
  - 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++)

Why study programming languages?

- Simulate useful features in languages that lack them
  - 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 suitable control structures in Fortran

    use comments and programmer discipline for control structures

Why study programming 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.

Why study programming languages?

- Help you choose a language
  - C vs. Modula-3 vs. C++ for systems programming
  - Fortran vs. APL vs. Ada for numerical computations
  - Ada vs. Modula-2 for embedded systems
  - Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  - Java vs. C/CORBA for networked PC programs

The Art of Language Design

What is a language for?

- Way of thinking – way of expressing algorithms
- Languages from the user's point of view
- Abstraction of virtual machine – way of specifying what you want the hardware to do without getting down into the bits
- Languages from the implementer's point of view

Why are there so many?

- Evolution - learn better ways of doing things over time
  - goto-based control flow (Fortran)
  - structured programming (Pascal, C)
  - object-oriented structure (C++, Java)
- Special purpose
  - symbolic data
  - character strings
  - low-level system programming
  - reasoning, logical relation
- Socio-economic factors - proprietary interests, commercial advantage
- Personal preference - diverse ideas about what is pleasant to use
- Special hardware

What makes a language successful?

- Expressive power – easy to express things, to use once fluent (C, APL, Algol-68, Perl)
- Ease of use for novice – easy to learn (BASIC, Pascal, LOGO)
- Ease of implementation – (BASIC, Forth)
- Standardization – (C, Java)
- Open source - wide dissemination without cost (Pascal, Java)
- Excellent compilers – possible to compile to very good (fast/small) code (Fortran)
- Patronage - backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)

Language Evaluation Criteria

Language Evaluation Criteria

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: conformance to specifications (i.e., performs to its specifications under all conditions)

Evaluation Criteria: Others

- Cost
  - the ultimate total cost
- Portability
  - the ease with which programs can be moved from one implementation to another
- Generality
  - the applicability to a wide range of applications
- Well-definedness
  - the completeness and precision of the language's official definition

Evaluation Criteria: Readability

- **Overall simplicity**
  - A manageable set of features and constructs
  - Few feature multiplicity (means of doing the same operation)
  - Minimal operator overloading
- **Orthogonality**
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
  - Lack of orthogonality leads to exceptions to rules
  - Makes the language easy to learn and read
  - Meaning is context independent

Evaluation Criteria: Readability

- **Control statements**
  - The presence of well-known control structures (e.g., `while` statement)
- **Data types and structures**
  - The presence of adequate facilities for defining data structures
- **Syntax considerations**
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- **Simplicity and orthogonality**
  - Few constructs, a small number of primitives, a small set of rules for combining them
- **Support for abstraction**
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
- **Expressivity**
  - A set of relatively convenient ways of specifying operations
  - Example: the inclusion of `for` statement in many modern languages

Evaluation Criteria: Reliability

- **Type checking**
  - Testing for type errors
- **Exception handling**
  - Intercept run-time errors and take corrective measures
- **Aliasing**
  - Presence of two or more distinct referencing methods for the same memory location
- **Readability and writability**
  - A language that does not support “natural” ways of expressing an algorithm will necessarily use “unnatural” approaches, and hence reduced reliability

Evaluation Criteria: Cost

- Training programmers to use language
- Writing programs
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Language Characteristics & Criteria

	Criteria		
Characteristic	Readability	Writability	Reliability
Simplicity & orthogonality	✓	✓	✓
Control statements	✓	✓	✓
Data types and structure	✓	✓	✓
Syntax design	✓	✓	✓
Support for abstraction		✓	✓
Expressivity		✓	✓
Type checking			✓
Exception handling			✓
Restricted aliasing			✓

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Conflicting criteria
  - Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
  - Another conflicting criteria
  - Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Another conflicting criteria
  - Example: C++ pointers are powerful and very flexible but not reliably used

Programming Language Paradigms

Programming Paradigms

- **Imperative**
  - Central features are variables, assignment statements, and iteration
  - Examples: C, Pascal
- **Object-oriented**
  - Data abstraction (Encapsulate data objects with processing), inheritance, dynamic type binding
  - Examples: Java, C++
- **Functional**
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- **Logic**
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- **Markup**
  - New; not a programming per se, but used to specify the layout of information in Web documents
  - Examples: XHTML, XML

Programming Paradigms

- Imperative
  - Von Neumann (Fortran, Pascal, Basic, C)
  - Scripting (Perl, Python, JavaScript, PHP)
  - Object-oriented (Smalltalk, Eiffel, C++)
- Declarative
  - Functional (Scheme, ML, pure Lisp, FP)
  - Logic, constraint-based (Prolog, VisiCalc, RPG)

Programming Paradigms: Alternatives

- Imperative
  - Procedural (C)
    - Block-Structured (Pascal, Ada)
  - Object-based (Ada)
    - Object-oriented (Ada, Object-Pascal, C++, Java)
  - Parallel Processing (Ada, Pascal-S, Occam, C-Linda)
- Declarative
  - Logic (Prolog)
  - Functional (LISP, Scheme)
  - Database (SQL)

Example of GCD program

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

```
(define gcd
   (lambda (a b)
     (cond ((= a b) a)
           ((> a b) (gcd (- a b) b))
           (else (gcd (- b a) a)))))

                                    ;scheme
```

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

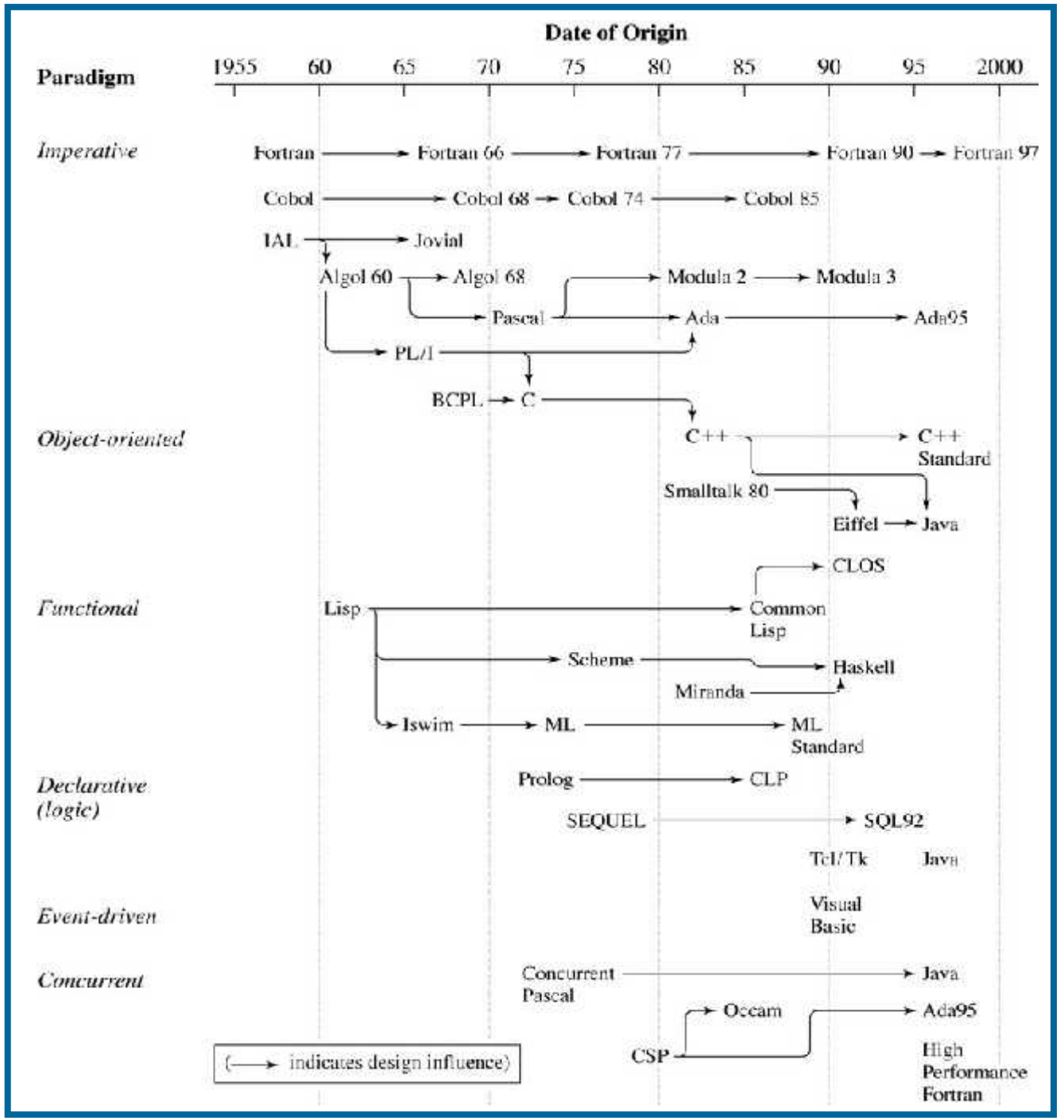
Programming Paradigms: Emerging

- Event-driven/Visual
  - Continuous loop that responds to events
  - Code is executed upon activation of events
  - Subcategory of imperative
  - Examples: Visual Basic .NET, Java
- Concurrent
  - Cooperating processes
  - Examples: High Performance Fortran

Programming Domains

- Scientific applications
  - Large number of floating point computations
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated
  - LISP
- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Programming
  - Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)

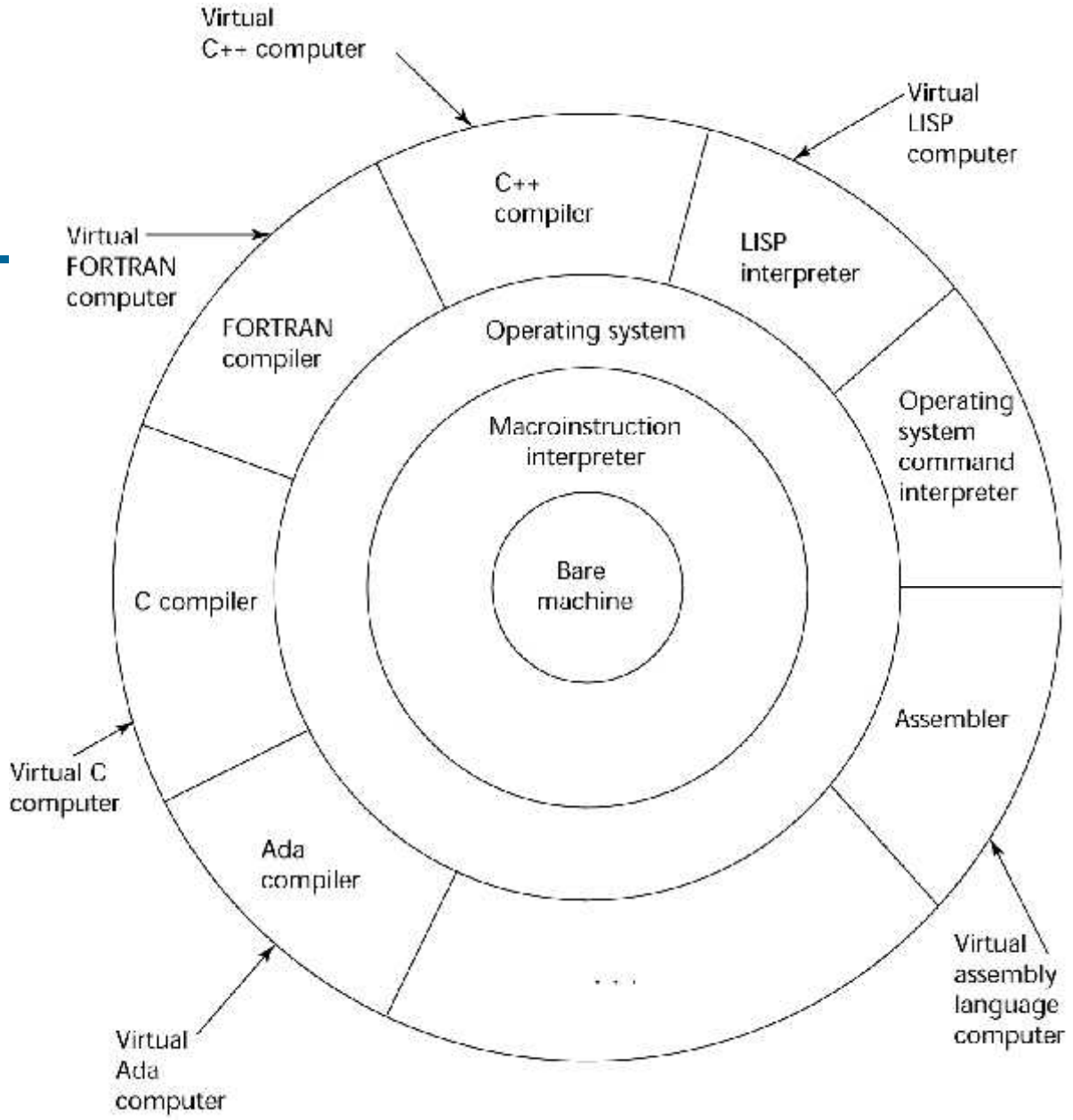
A Brief Historical Lineage of Some Key Programming Languages



Implementation Methods

Layered View of Computer

Virtual computer – the OS and language implementation which are layered over machine interface of a computer



Implementation Methods

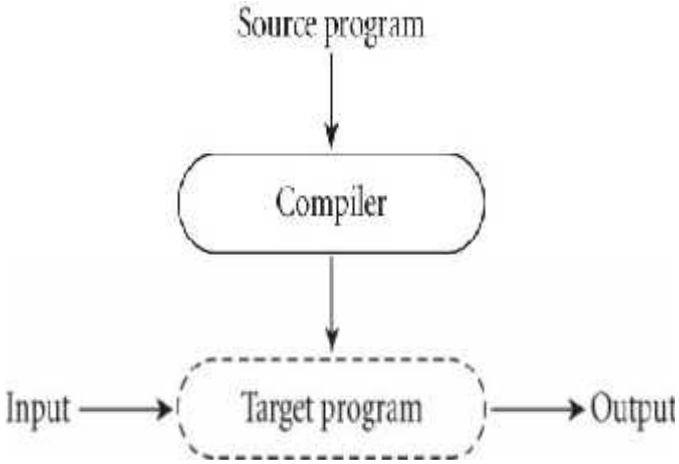
- **Compilation**
  - Programs are translated into machine language
- **Pure Interpretation**
  - Programs are interpreted by another program known as an interpreter
- **Hybrid /Mixing**
  - A compromise between compilers and pure interpreters

Implementation Methods

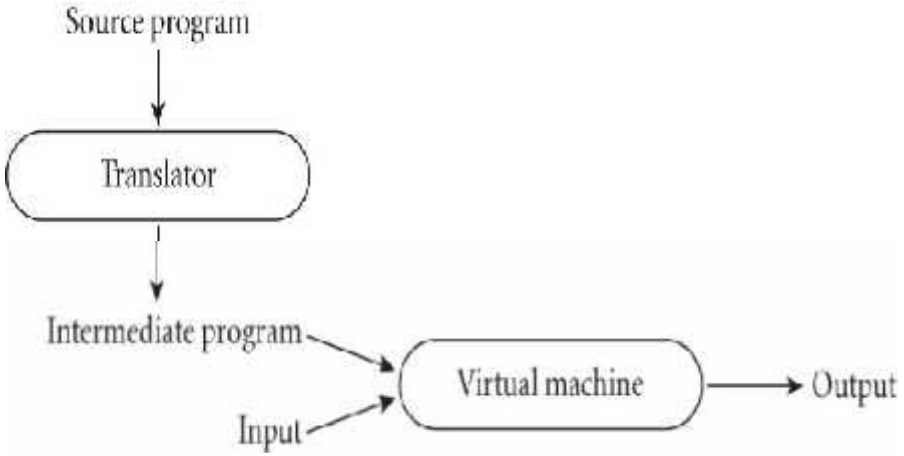
- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)
- Compilation
  - Better performance

Implementation Methods

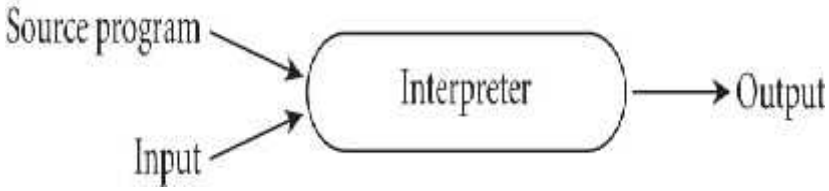
- Compilation

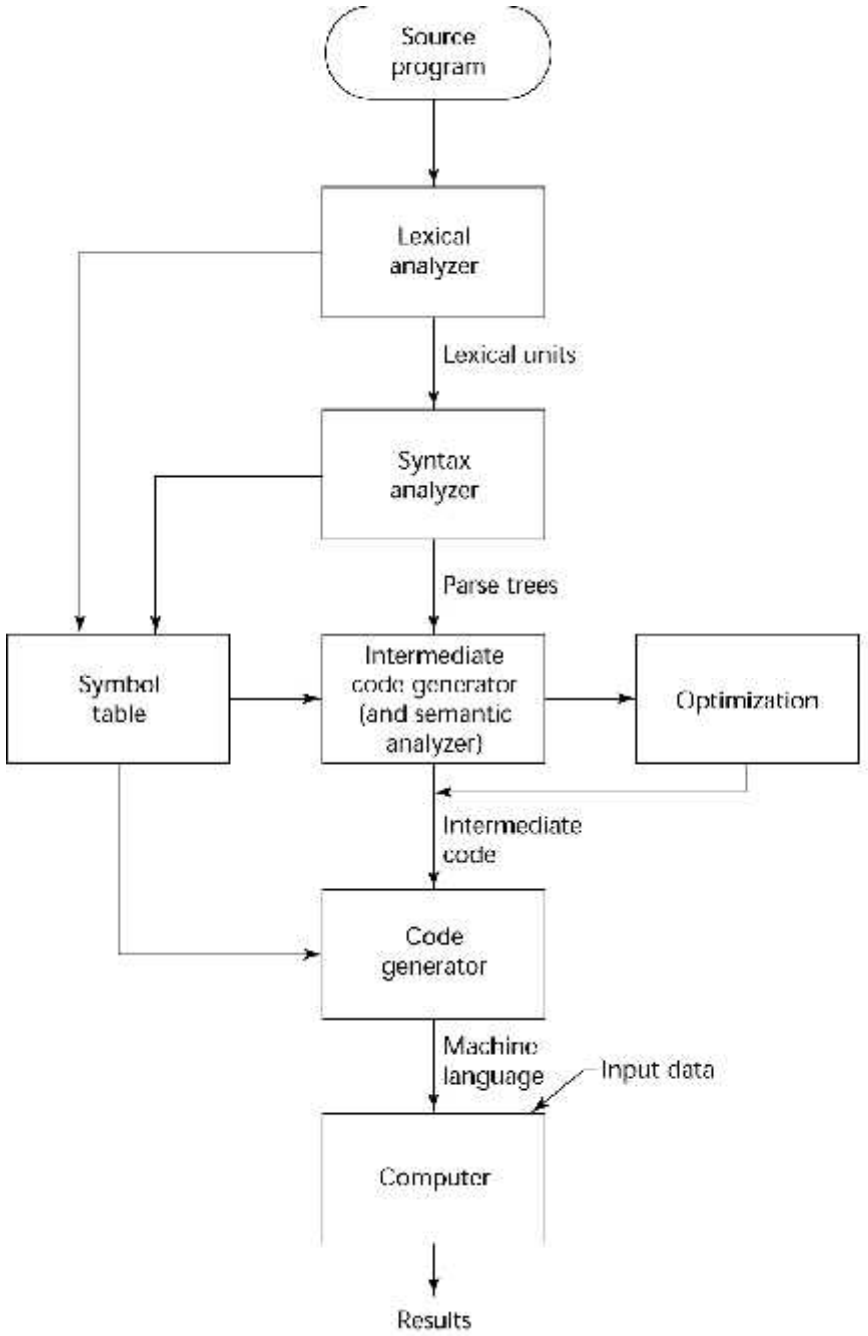


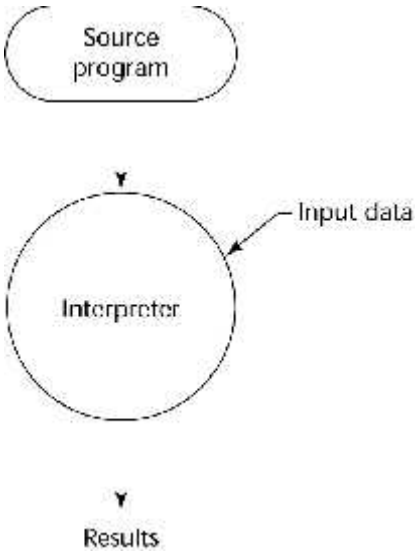
- Mixing

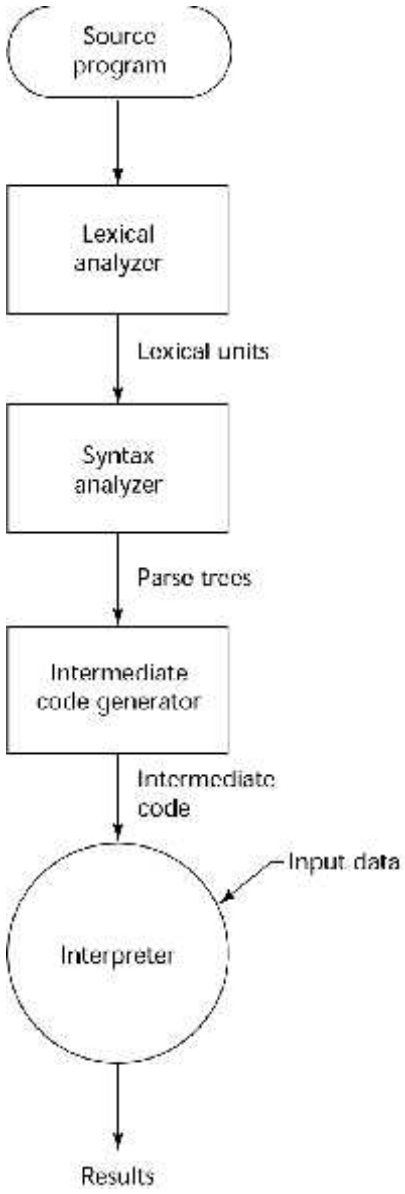


- Interpretation



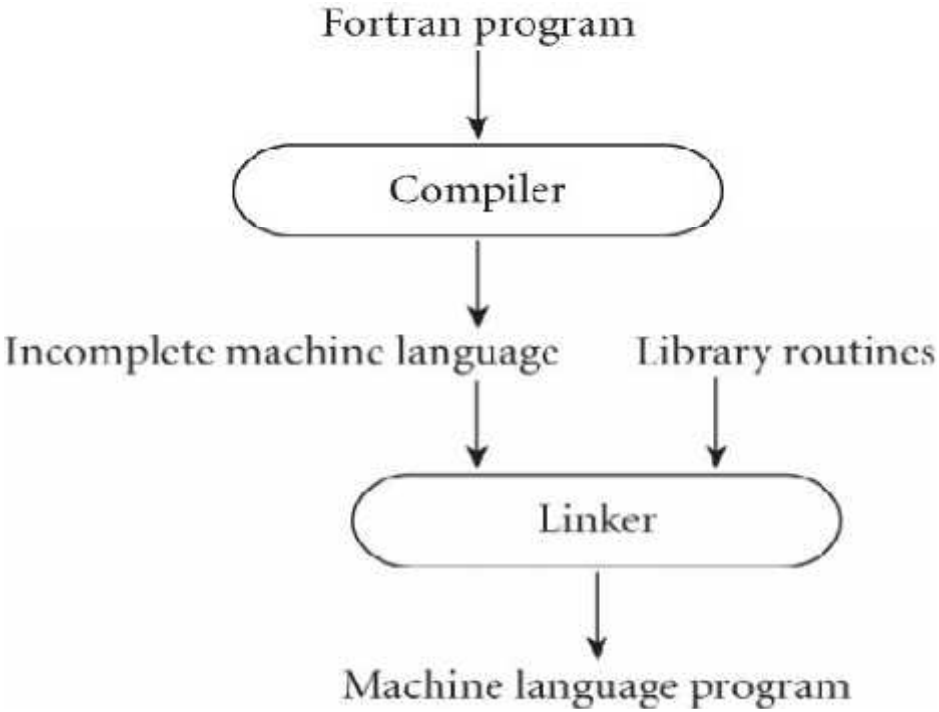

Source program
Lexical analyzer
Lexical units
Syntax analyzer
Parse trees
Symbol table
Intermediate code generator (and semantic analyzer)
Optimization
(optional)
Intermediate code
Code generator
Machine language
Input data
Computer
Results


Source program
Input data
Interpreter
Results

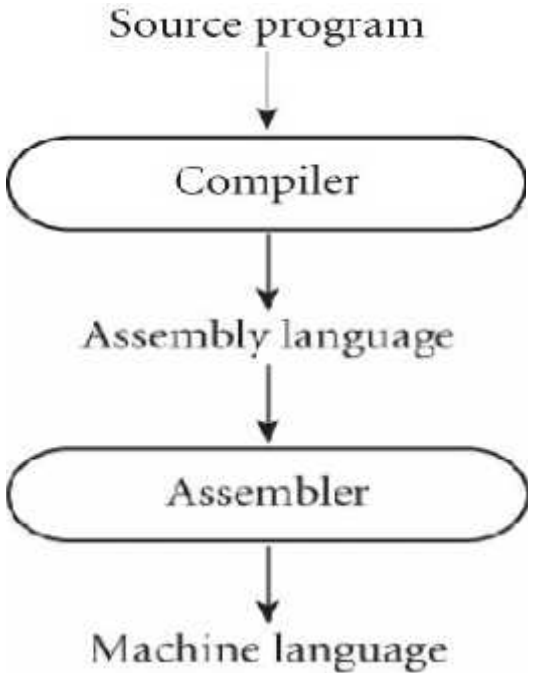

Source program
Lexical analyzer
Lexical units
Syntax analyzer
Parse trees
Intermediate code generator
Intermediate code
Input data
Interpreter
Results

Implementation Methods

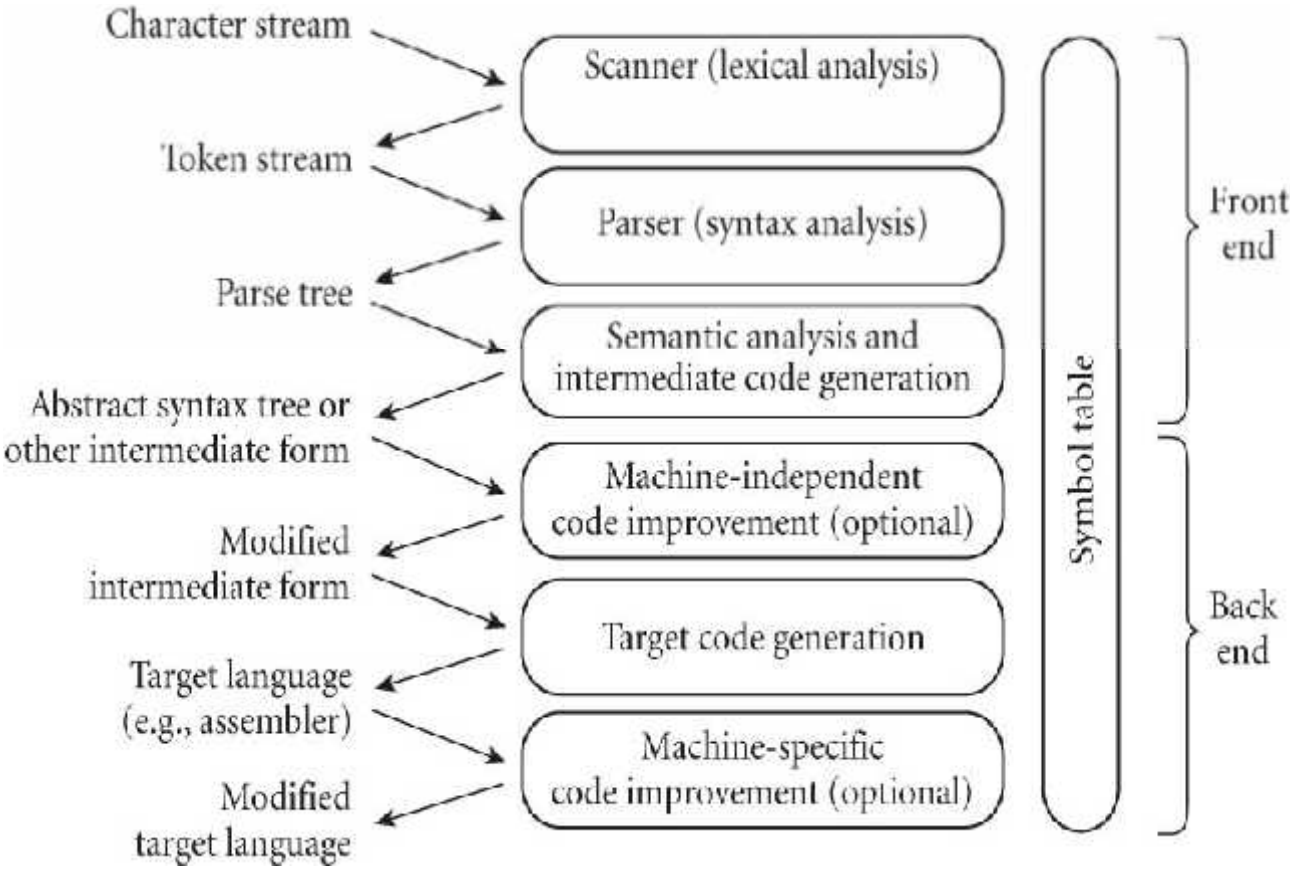
- Library routines and linking



- Post-compilation assembly



Overview of Compilation



Overview of Compilation

- Lexical analysis
- Syntax analysis
- Semantic Analysis & intermediate code generation
- Target code generation
- Code improvement

Overview of Compilation

- 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);
}
```

Overview of Compilation

- Lexical Analysis
  - Scanning and parsing recognize the structure of the program, groups characters into *tokens*, the smallest meaningful units of the program

GCD Program Tokens

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

Overview of Compilation

- 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

Overview of Compilation

- Context-Free Grammar and Parsing

  Example (`while` loop in C)

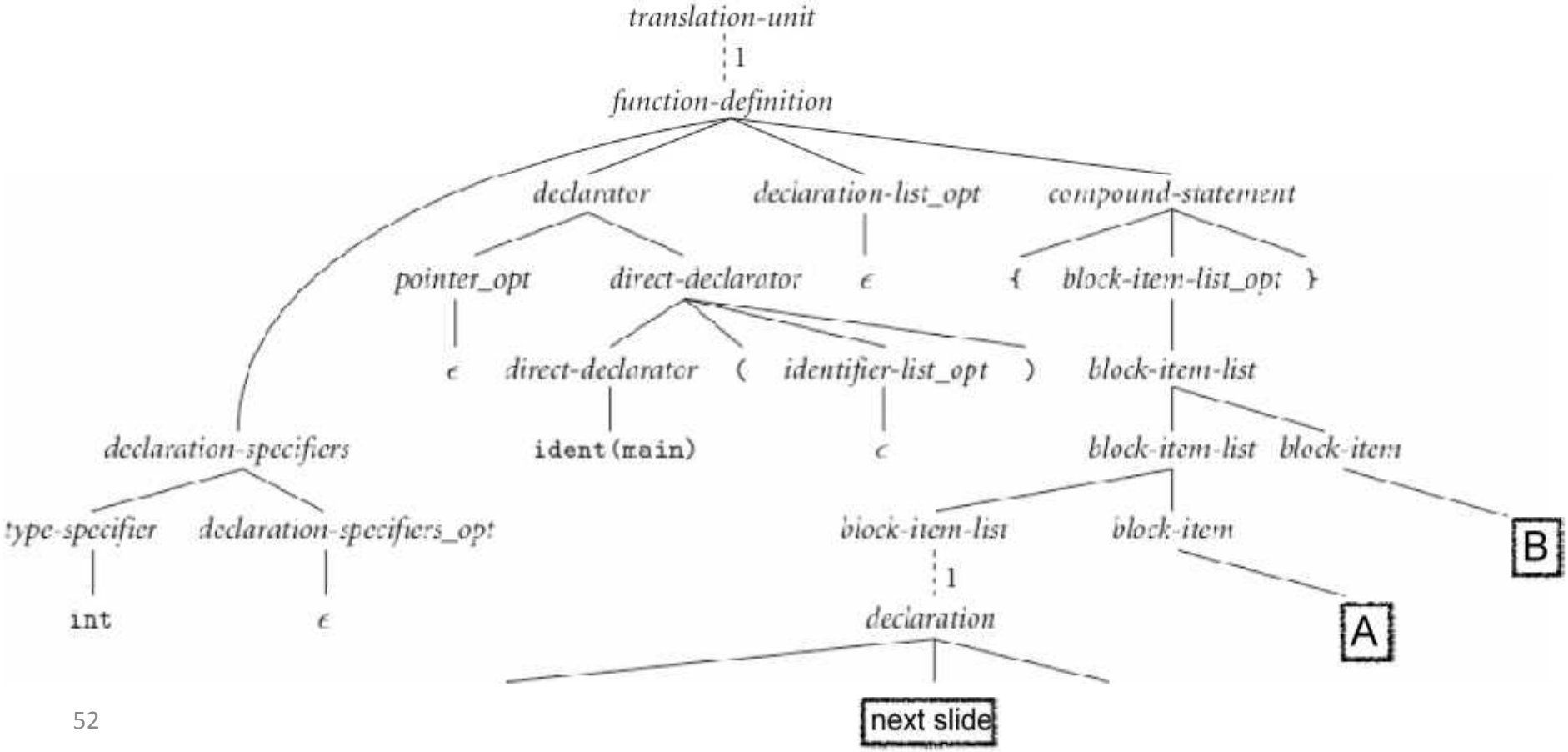
  iteration-statement → while (expression) statement

  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

Overview of Compilation

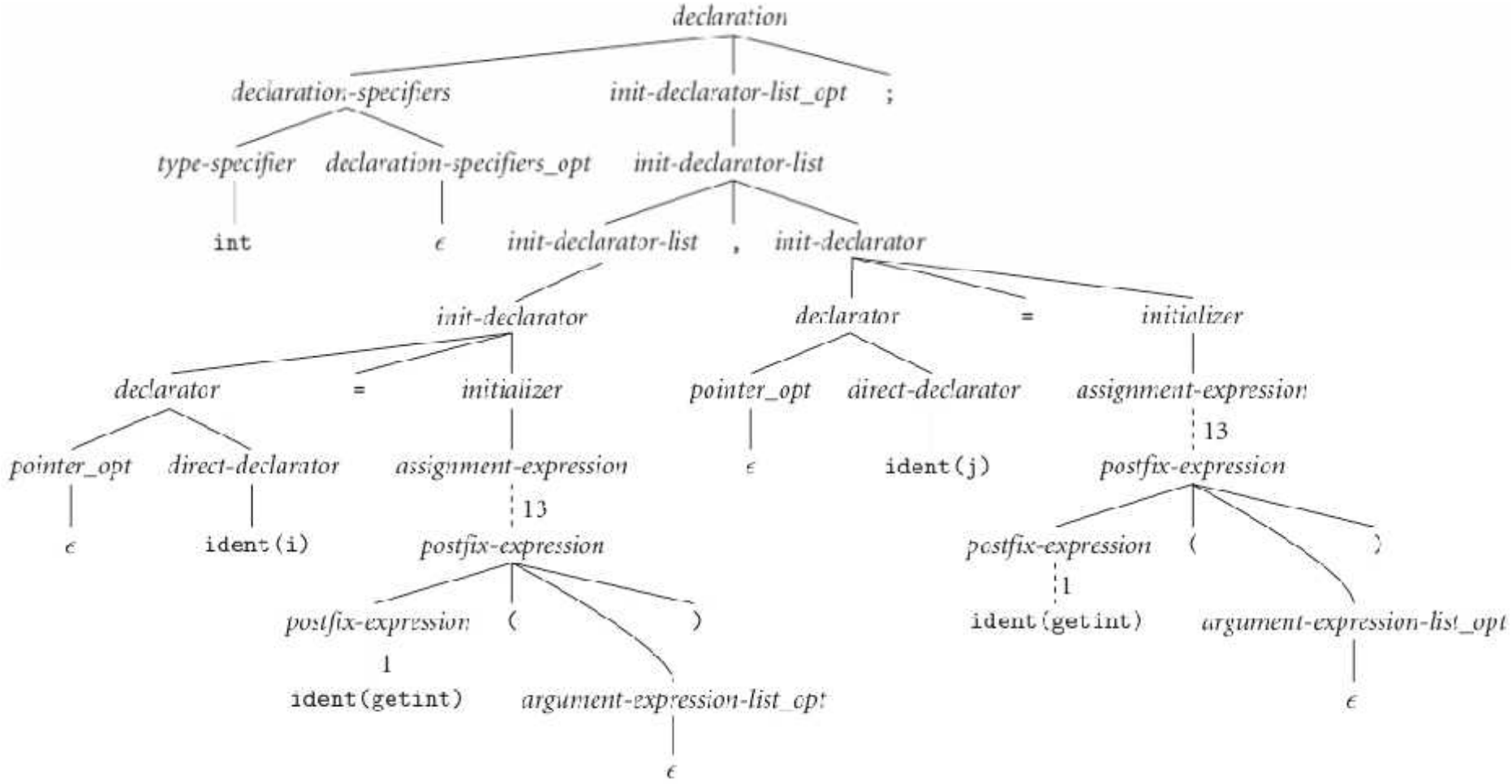
- Context-Free Grammar and Parsing

  GCD Program Parse Tree



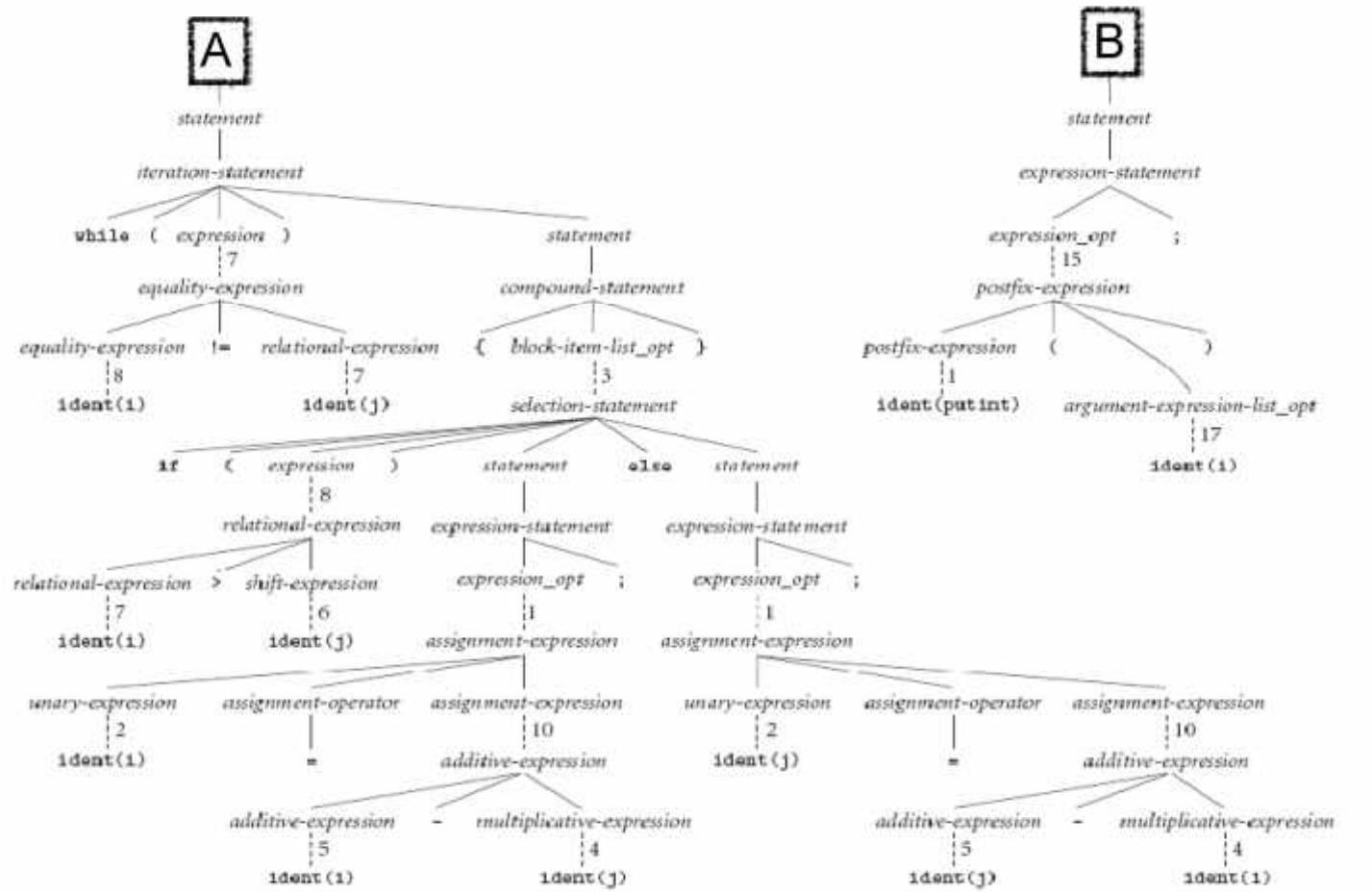
Overview of Compilation

- Context-Free Grammar and Parsing



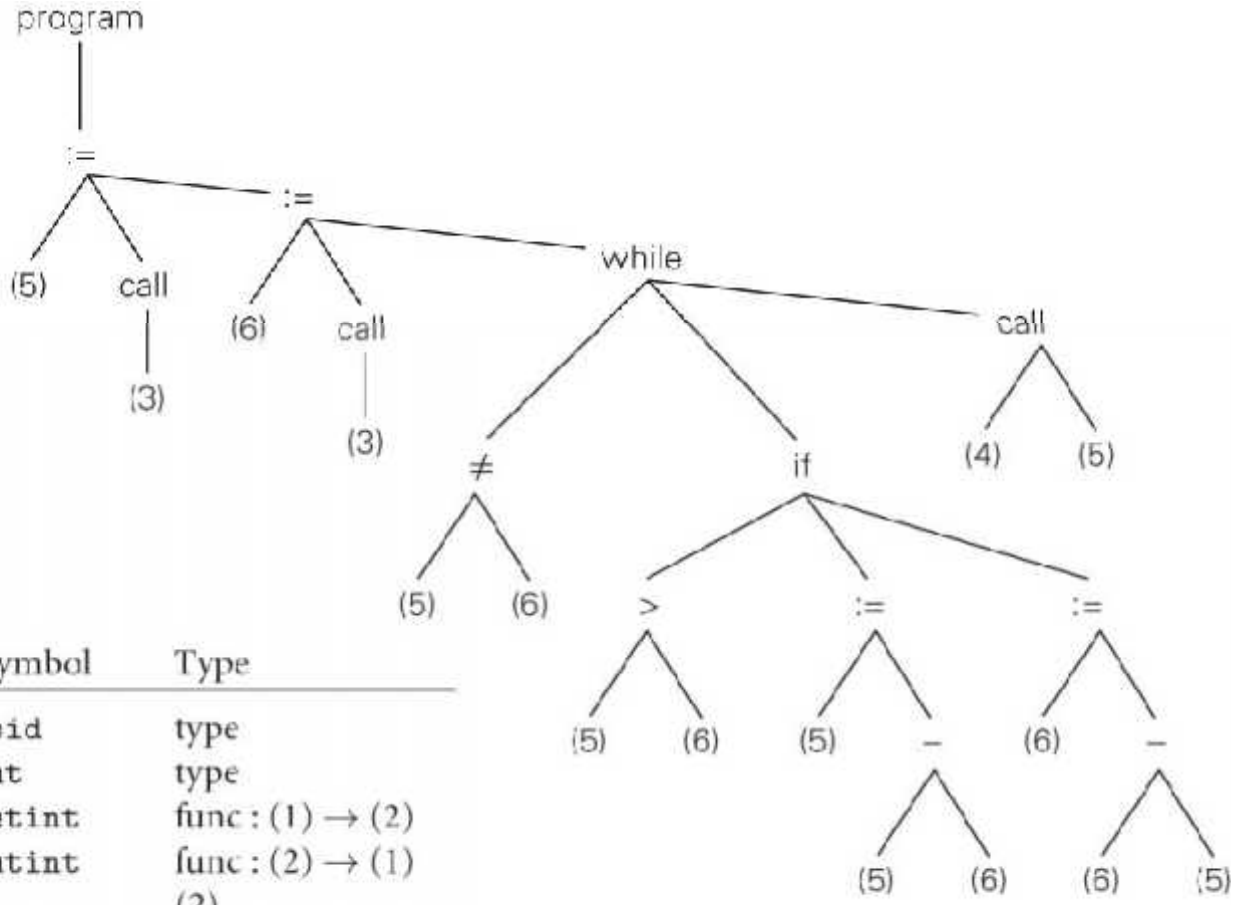
Overview of Compilation

- Context-Free Grammar and Parsing



Overview of Compilation

- 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)

Programming Environments

Programming Environments

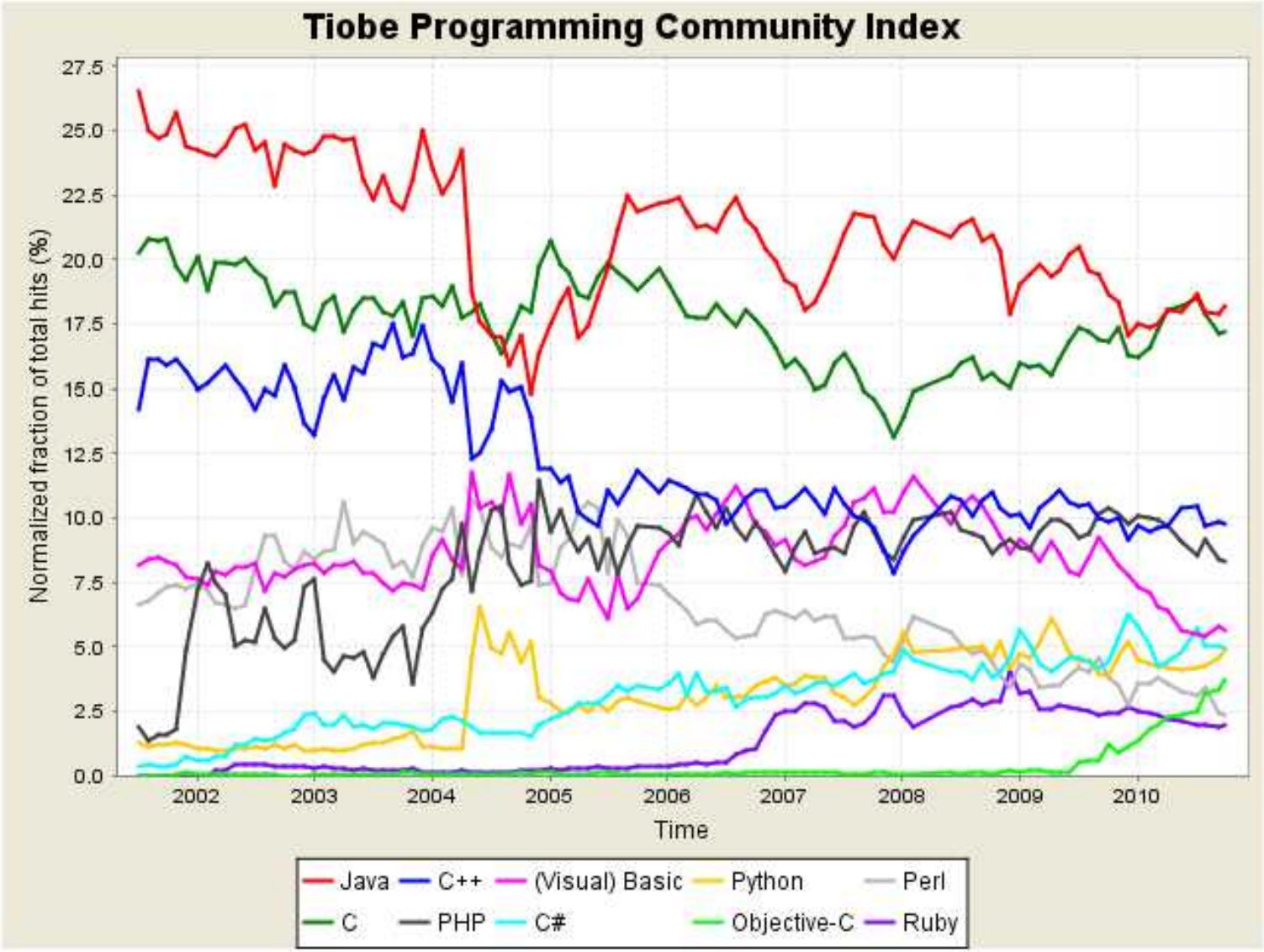
- The collection of tools used in software development
- UNIX
  - An older operating system and tool collection
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that run on top of UNIX
- Borland JBuilder
  - An integrated development environment for Java
- Microsoft Visual Studio.NET
  - A large, complex visual environment
  - Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

Programming Environments

Tools

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`

Programming Languages: Trend



Programming Languages: Trend

Position Oct 2010	Position Oct 2009	Delta in Position	Programming Language	Ratings Oct 2010	Delta Oct 2009	Status
1	1	=	Java	18.166%	-0.48%	A
2	2	=	C	17.177%	+0.33%	A
3	4	↑	C++	9.802%	-0.08%	A
4	3	↓	PHP	8.323%	-2.03%	A
5	5	=	(Visual) Basic	5.650%	-3.04%	A
6	6	=	C#	4.963%	+0.55%	A
7	7	=	Python	4.860%	+0.96%	A
8	12	↑↑↑↑	Objective-C	3.706%	+2.54%	A
9	8	↓	Perl	2.310%	-1.45%	A
10	10	=	Ruby	1.941%	-0.51%	A
11	9	↓↓	JavaScript	1.659%	-1.37%	A
12	11	↓	Delphi	1.558%	-0.58%	A
13	17	↑↑↑↑	Lisp	1.084%	+0.48%	A-
14	24	↑↑↑↑↑↑↑↑↑↑	Transact-SQL	0.820%	+0.42%	A-
15	15	=	Pascal	0.771%	+0.10%	A-
16	18	↑↑	RPG (OS/400)	0.708%	+0.12%	A-
17	29	↑↑↑↑↑↑↑↑↑↑↑↑	Ada	0.704%	+0.40%	A--
18	14	↓↓↓↓	SAS	0.664%	-0.14%	B
19	19	=	MATLAB	0.627%	+0.05%	B
20	-	↑↑↑↑↑↑↑↑↑↑	Go	0.626%	+0.63%	B

References

Books

- Robert W. Sebesta. *Concepts of Programming Languages*, Addison Wesley, 2006.
- Michael L. Scott. *Programming Language Pragmatics.* Morgan Kaufmann Publishers, 2009

Interesting links

- history:http://en.wikipedia.org/wiki/History_of_programming_languages
- timeline: http://www.levenez.com/lang/lang_a4.pdf