Introduction

Chapter 1

Programming languages - ubiquitous













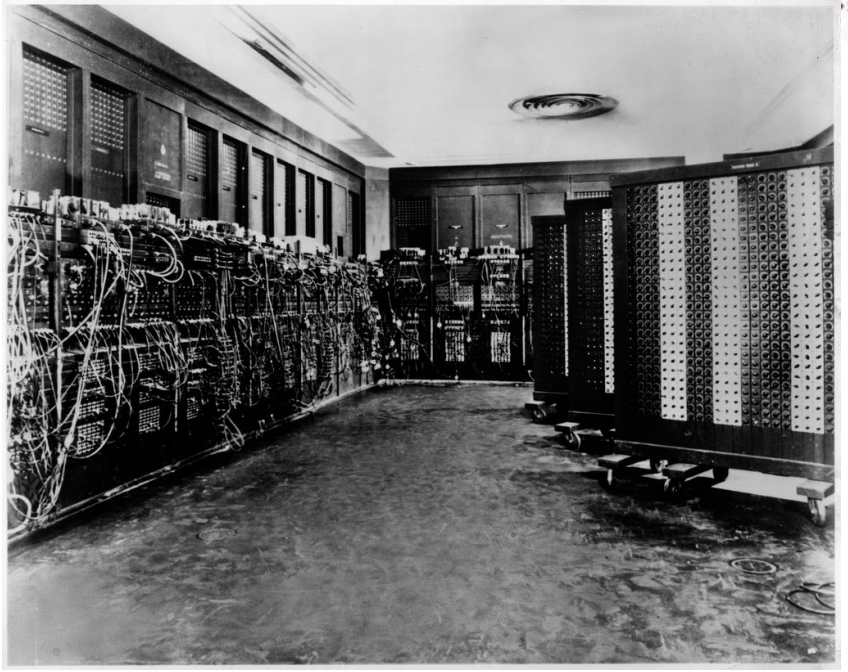






Computer evolution

- ENIAC
  - 18,000 sq feet
  - 25 tones = 25,000 Kg
  - 5,000 instr/s



- iPhone 6
  - 4.55 ounces = 0.13 Kg
  - 25,000,000,000 instr/s

  - 200,000 x smaller, 5,000,000 x faster
    = 1,000,000,000,000 x more efficient

Computer evolution - Quotes

- *"I think there is a world market for maybe five computers."*

  (Thomas Watson, president of IBM, 1943)

- *"Where a calculator like the ENIAC today is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh only 1½ tons."*

  (Andrew Hamilton, "Brains that Click", 1949)

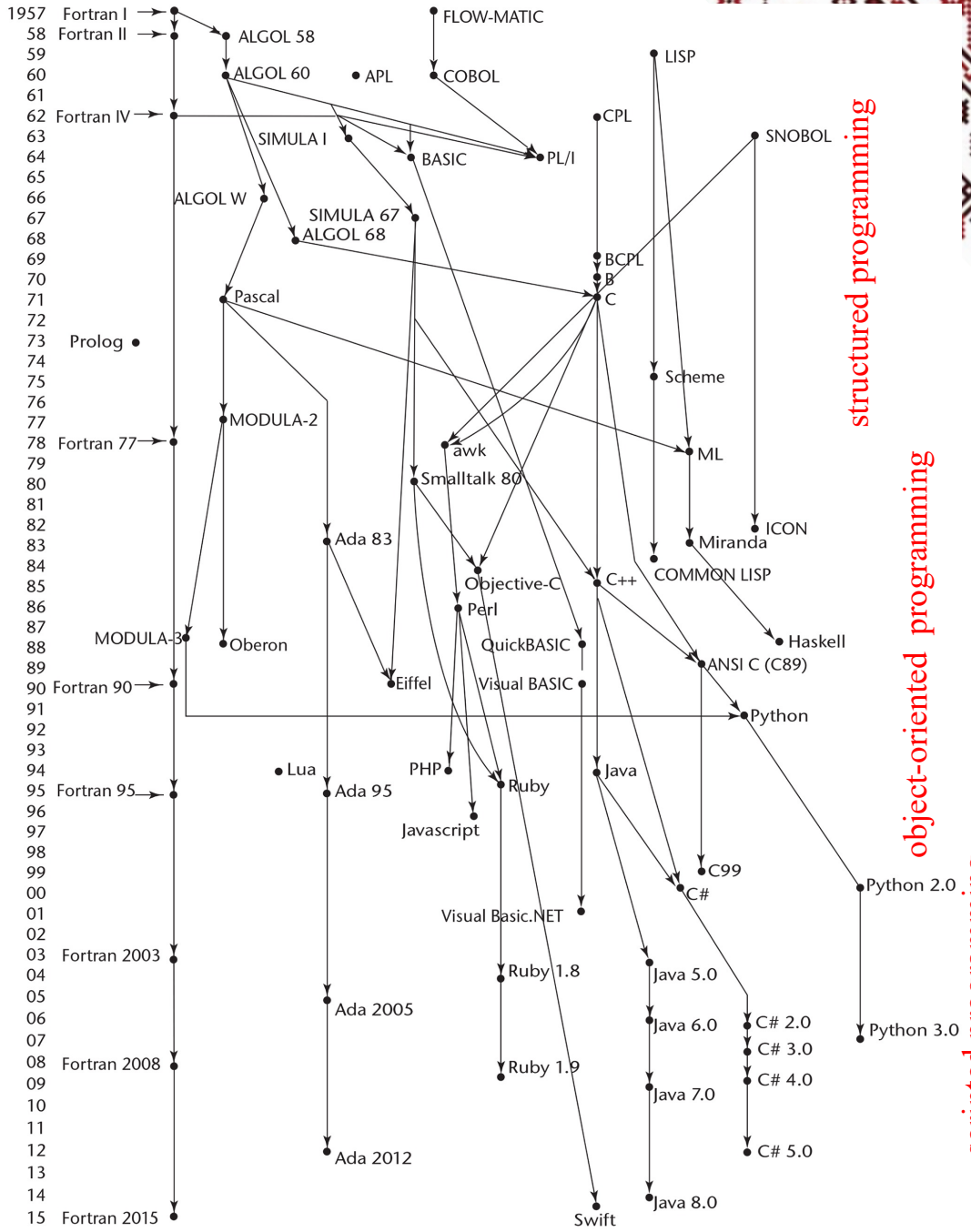
- *"The cost for 128 kilobytes of memory will fall below U$100 in the near future."*

  (Creative Computing magazine, December 1981)

Introduction

Why are there so many languages?

- Evolution
- Special purposes
- Personal preference
- Features
- Availability
  - Standardization
  - Open source
- Good compilers
- Socio-economic factors



Introduction

What are programming languages for?

- way of thinking - expressing algorithms
- abstraction of virtual machine - way of specifying what you want the hardware to do without getting down into the bits
- implementor's point of view *vs.* programmer's point of view

> *"Programming is the art of telling another human being what one wants the computer to do."*
>
> Donald Knuth

- conceptual clarity
- implementation efficiency

Introduction

What makes a language successful?

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

Programming languages spectrum

- imperative – how the computer should do it?
  - von Neumann - C, Fortran, Pascal, Basic
  - object-oriented - C++, Smalltalk, Java
  - scripting languages - Python, Perl, JavaScript, PHP
- declarative – what the computer is to do?
  - functional - Scheme, ML, Lisp, FP
  - logic - Prolog, VisiCalc, RPG

- imperative languages predominate
  - better performance
- declarative languages are higher level
  - farther from implementation details
  - safer; imperative languages started importing their features

Evolution

- Machine language

```
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
```

- Assembly

```
    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  putint
    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
```

- Fortran

```
FUNCTION GCD(A, B)
        IA = A
        IB = B
   1    IF (IB.NE.0) THEN
          ITEMP = IA
          IA = IB
          IB = MOD(ITEMP, IB)
          GOTO 1
        END IF
        GCD = IA
        RETURN
      END
```

Evolution

- C++

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

```
int gcd2(int a, int b) {
    return (b==0) ? a : gcd2(b, a%b);
}
```

- Python

```
def gcd(x, y):
    while (y):
        x, y = y, x % y
    return x


def gcd2(a,b):
    return a if (b==0) else gcd2(b, a%b)
```

Evolution

- Scheme

```
(define gcd
  (lambda (a b)
    (cond ((zero? b) a)
          (else (gcd b (modulo a b))))))
```

- Prolog

```
gcd(X,Y,G) :- X=Y, G=X.
gcd(X,Y,G) :- X<Y, Y1 is Y-X, gcd(X,Y1,G).
gcd(X,Y,G) :- X>Y, gcd(Y,X,G).
```

Why study programming languages?

- Help you choose a language:
  - systems programming: C, C++, C#
  - numerical computations: Fortran, C, Matlab
  - web-based applications: PHP, Javascript, Ruby
  - embedded systems: Ada, C
  - symbolic data manipulation: Scheme, ML, Common Lisp
  - networked PC programs: Java, .NET
  - logical relationships: Prolog
- Make it easier to learn new languages:
  - many concepts are common to many languages: syntax, semantics, iteration, recursion, abstraction, etc.
- Make better use of the language you are using:
  - understand various features
  - understand implementation cost
  - find ways to do things that are not explicitly supported

Top Languages

TOP 10

Popular Programming Languages in 2020

1	Python
2	JavaScript
3	Java
4	C#
5	C
6	C++
7	GO
8	R
9	Swift
10	PHP

Our List of the Top 20 Programming Languages

1. JavaScript (React.js and Node.js)
2. Python
3. HTML
4. CSS
5. C++
6. TypeScript
7. Rust
8. Scheme
9. Java
10. Kotlin
11. C#
12. Perl
13. PHP
14. Scala
15. Swift
16. MATLAB
17. SQL
18. R Programming Language
19. Golang (Go)
20. Ruby

Top 10 Most Popular Programming Languages In 2020

Oct 2020	Programming Language	Ratings
1	C	16.95%
2	Java	12.56%
3	Python	11.28%
4	C++	6.94%
5	C#	4.16%
6	Visual Basic	3.97%
7	JavaScript	2.14%
8	PHP	2.09%
9	R	1.99%
10	SQL	1.57%

The Power of Abstraction

- Abstraction - ability to control complexity
  - high-level programming
  - names
  - functions / procedures / methods
  - objects
  - functional programming

  - *"Mathematics is the queen of the sciences."*

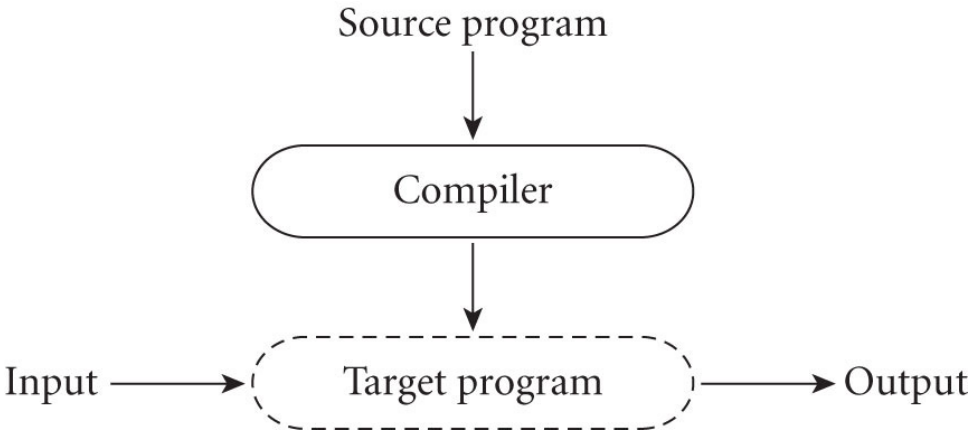
    Carl Friedrich Gauss

  - *"Mathematics is the language with which God has written the universe."*

    Galileo Galilei

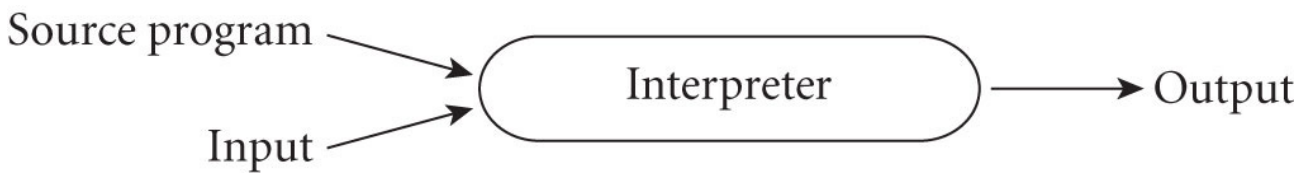
Compilation vs. Interpretation

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



Compilation vs. Interpretation

- Interpretation
  - Interpreter stays around for the execution of the program
  - Interpreter is the locus of control during execution

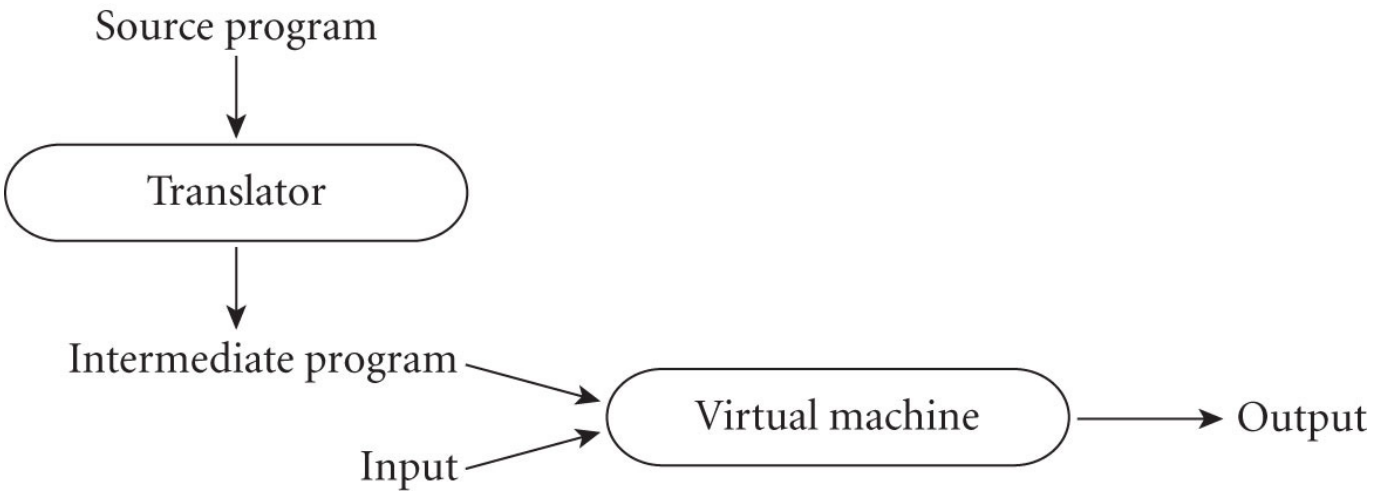


Compilation vs. Interpretation

- Compilation
  - Better performance
    - Early decisions can save time (*early vs. late binding*)
    - Example: a variable's address can be fixed at compile time
- Interpretation:
  - Greater flexibility
    - Example: Lisp, Prolog programs can write new pieces and execute them on the fly
  - Better diagnostics - error messages
    - Source-level debugger

Compilation vs. Interpretation

- Compilation, then interpretation
  - Distinction not very clear; compiled if:
    - Translator analyzes the program thoroughly
    - Intermediate program very different from source
  - Python – interpreted: dynamic semantic error checking
  - C, Fortran – compiled: static semantic error checking



Compilation vs. Interpretation

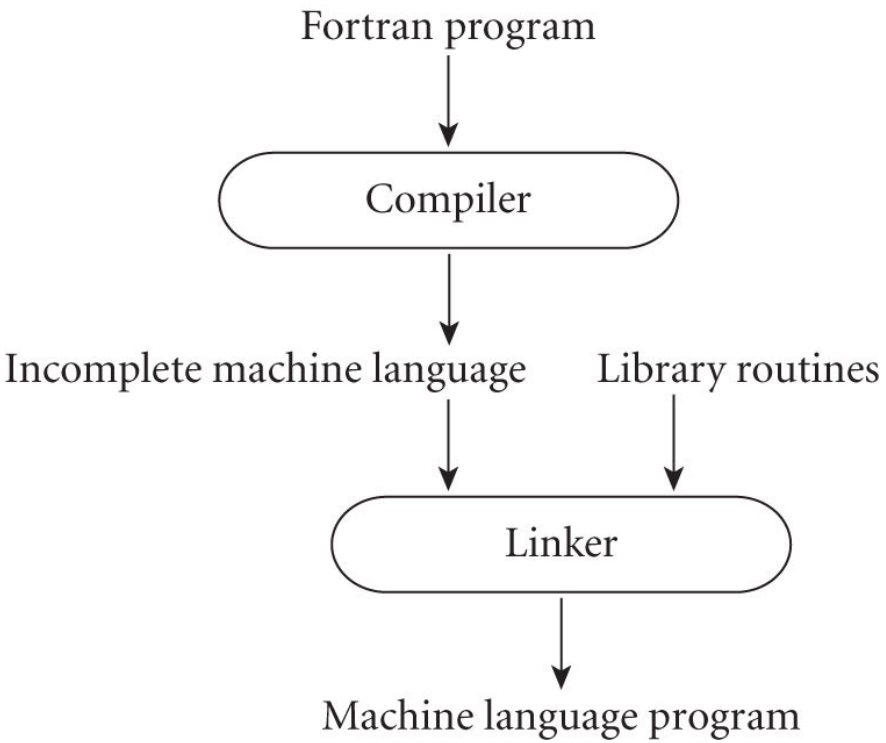
- Compilation
  - Compilation is *translation* from one language into another, with full analysis of the meaning of the input
  - Compilation entails semantic *understanding* of what is being processed; pre-processing does not
  - A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

Implementation strategies

- Preprocessor
  - Used by many interpreted languages
  - 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)

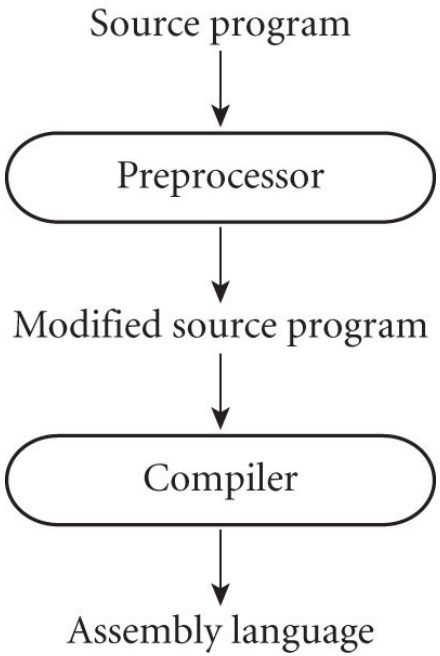
Implementation strategies

- 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:



Implementation strategies

- The C Preprocessor (conditional compilation)
  - Preprocessor deletes comments and expands macros
  - Preprocessor deletes portions of code, which allows several versions of a program be built from same source
    - Example: `#ifdef` directive

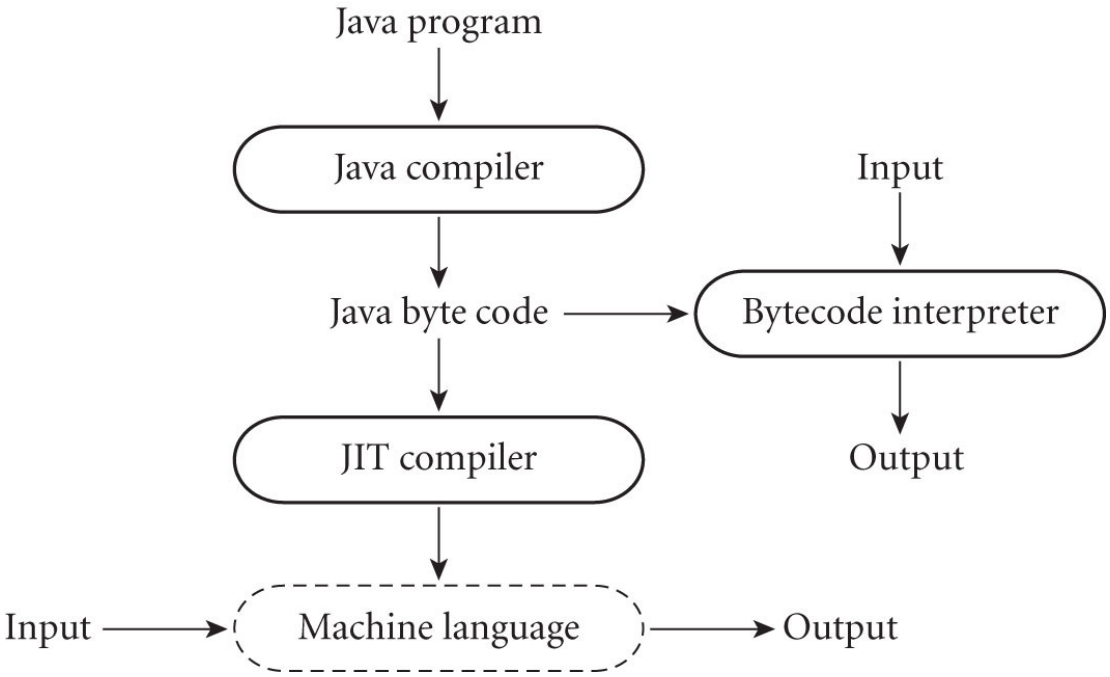


Implementation strategies

- Compilation of Interpreted Languages
  - Interpreted/compiled is a property of the implementation, not of the language
  - Python, Lisp, Prolog, Smalltalk
  - 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.

Implementation strategies

- Just-in-Time Compilation
  - Delay compilation until the last possible moment
    - Java: machine-independent intermediate form – bytecode
      - bytecode is the standard format for distribution of Java programs
    - C# compiler produces Common Intermediate Language (CIL)



Implementation strategies

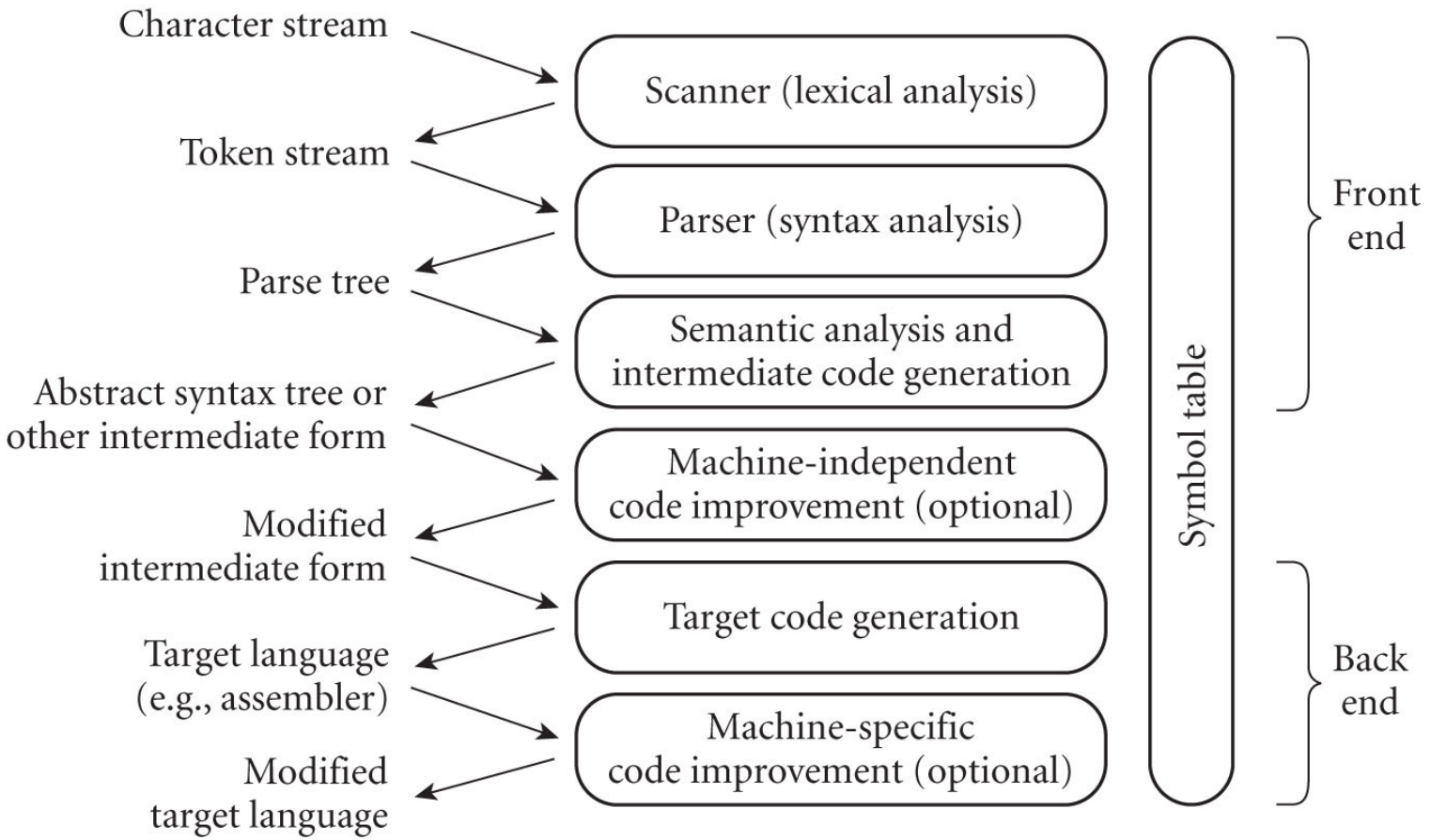
- Unconventional compilers
  - text formatters may compile high-level document description into commands for a printer
    - TEX, LATEX
  - query language processors translate into primitive operations on files
    - SQL

Programming Environment Tools

- Tools
  - Assemblers, debuggers, preprocessors, linkers
  - Editors – can have cross referencing
  - Version management – keep track of separately compiled modules
  - Profilers – performance analysis
  - IDEs – help with everything
    - knowledge of syntax
    - maintain partially compiled internal representation
    - Eclipse, NetBeans, Visual Studio, XCode

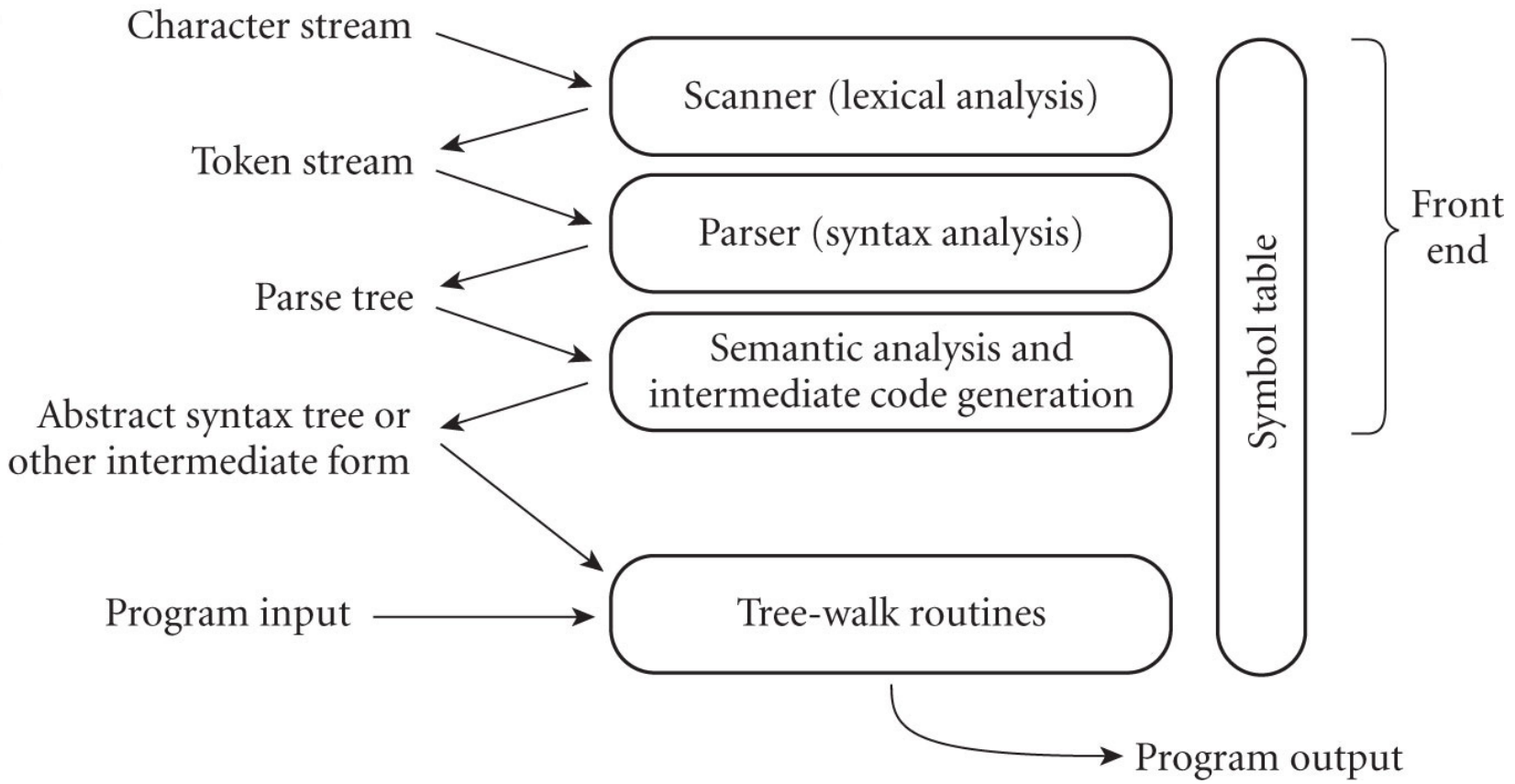
An Overview of Compilation

- Phases of Compilation



An Overview of Interpretation

- Phases of Interpretation



An Overview of Compilation

- Scanning (Lexical Analysis)
  - divide program into "tokens"
    - smallest meaningful units
    - this saves time, since character-by-character processing is slow
  - scanning is recognition of a regular language
    - via a DFA (Deterministic Finite Automaton)

An Overview of Compilation

- Scanning: Example
  - C Program (computes GCD):

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

  - Input – sequence of characters:
    - ‘i’, ‘n’, ‘t’, ‘ ’, ‘m’, ‘a’, ‘i’, ‘n’, ‘(’, ‘)’, …
  - Output – tokens:
    - `int, main, (, ), {, int, i, =, getint, (, ), j, =, getint, (, ), ;, while, (, i, !=, j, ), {, if, (, i, >, j, ), i, =, i, -, j, ;, else, j, =, j, -, i, ;, }, putint, (, i, ), ;, }`

An Overview of Compilation

- Parsing (Syntax Analysis)
  - discovers the structure of the program
  - parsing is recognition of a context-free language
    - via a Push-Down Automaton (PDA)
  - organize tokens into a parse tree
    - higher-level constructs in terms of their constituents
    - as defined by a context-free grammar

An Overview of Compilation

- Parsing: Example – `while` loop in C
  - Context-free grammar (part of):

iteration-statement → `while (` *expression* `)` *statement*

statement → `{` *block-item-list-opt* `}`

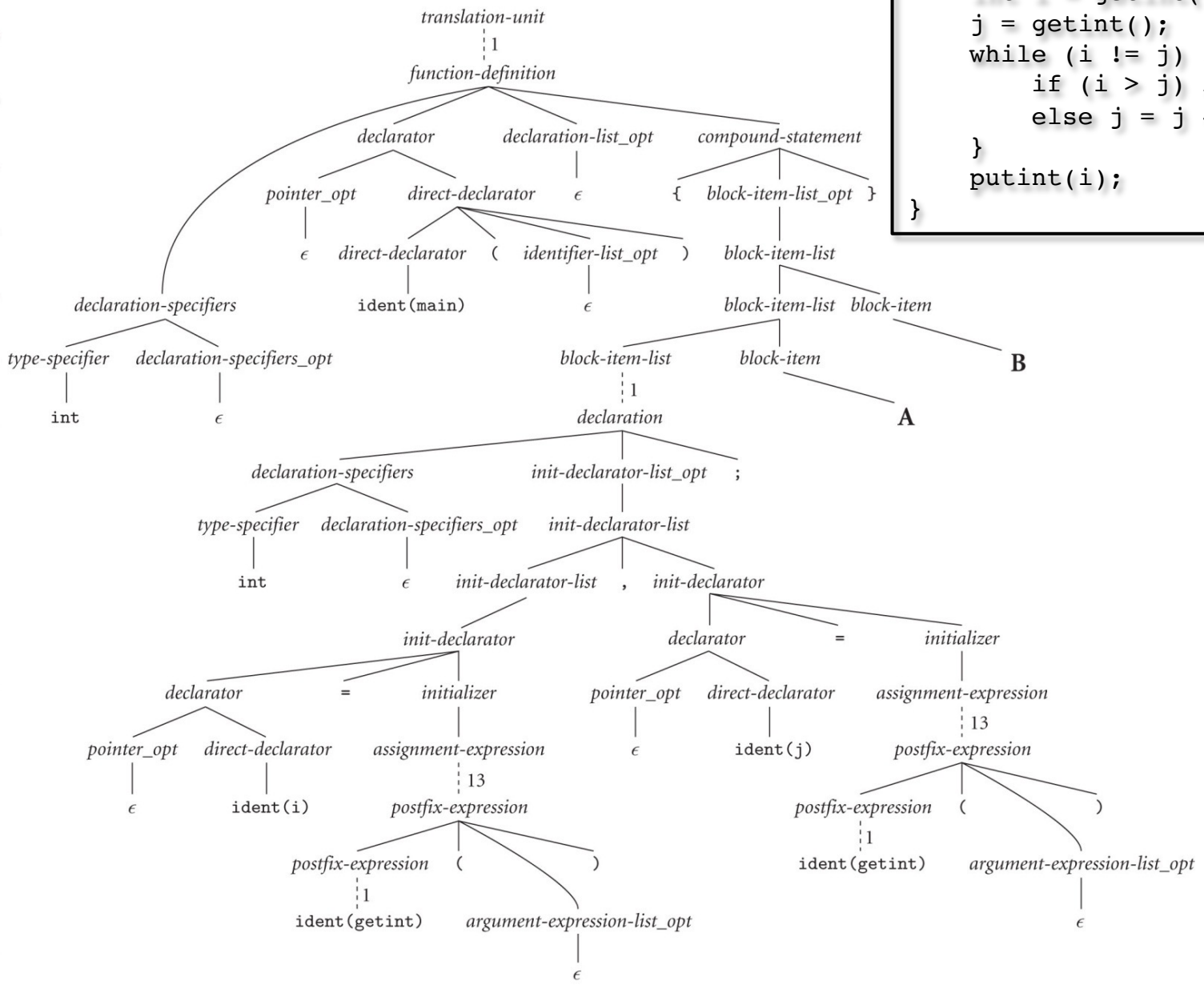
block-item-list-opt → *block-item-list* | ϵ

block-item-list → *block-item*
block-item-list → *block-item-list block-item*

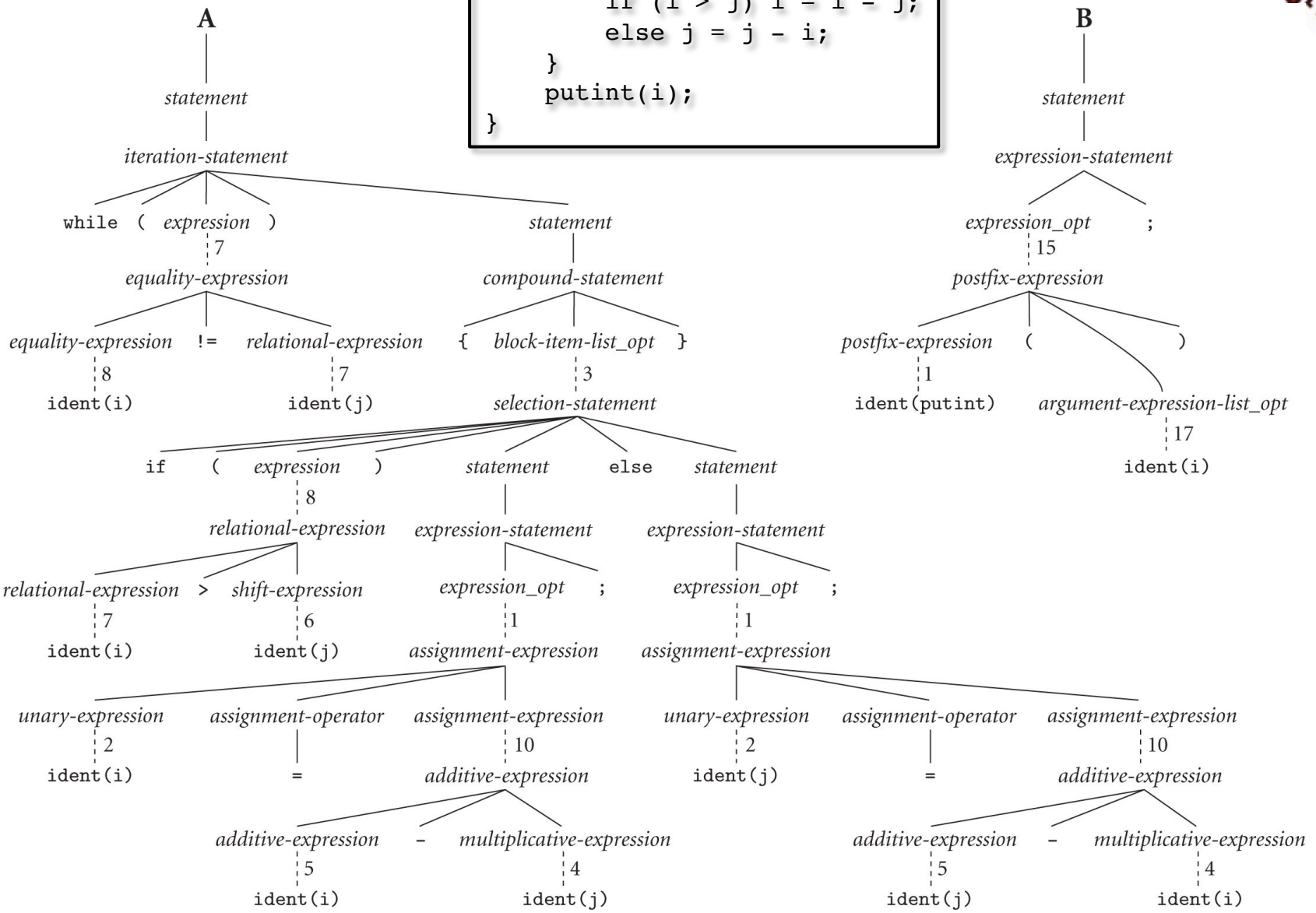
block-item → *declaration*
block-item → *statement*

  - Parse tree for GCD program
    - based on full context-free grammar
    - see next slides

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



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



An Overview of Compilation

- Semantic Analysis
  - the discovery of meaning in the program
  - detects multiple occurrences of the same identifier
  - tracks the *types* of identifiers and expressions
  - verify consistent usage and guide code generation
  - builds and maintains a *symbol table:*
    - maps each identifier to its information: type, scope, structure, etc.
    - used to check many things
    - Examples in C:
      - identifiers declared before used
      - identifiers used in the appropriate context
      - correct number and type of arguments for subroutines
      - return correct type
      - switch arms have distinct constant labels

An Overview of Compilation

- Semantic Analysis
  - compiler does *static* semantic analysis
  - *dynamic* semantics - for what must be checked at run time
  - Dynamic checks - trade off: safety vs. speed
    - C has very few dynamic checks
  - Examples in other languages:
    - array indexes within bounds
    - variables initialized before used
    - pointers are dereferenced only when referring to valid object
    - arithmetic operations do not overflow
  - Run time checks fail – abort or throw exception

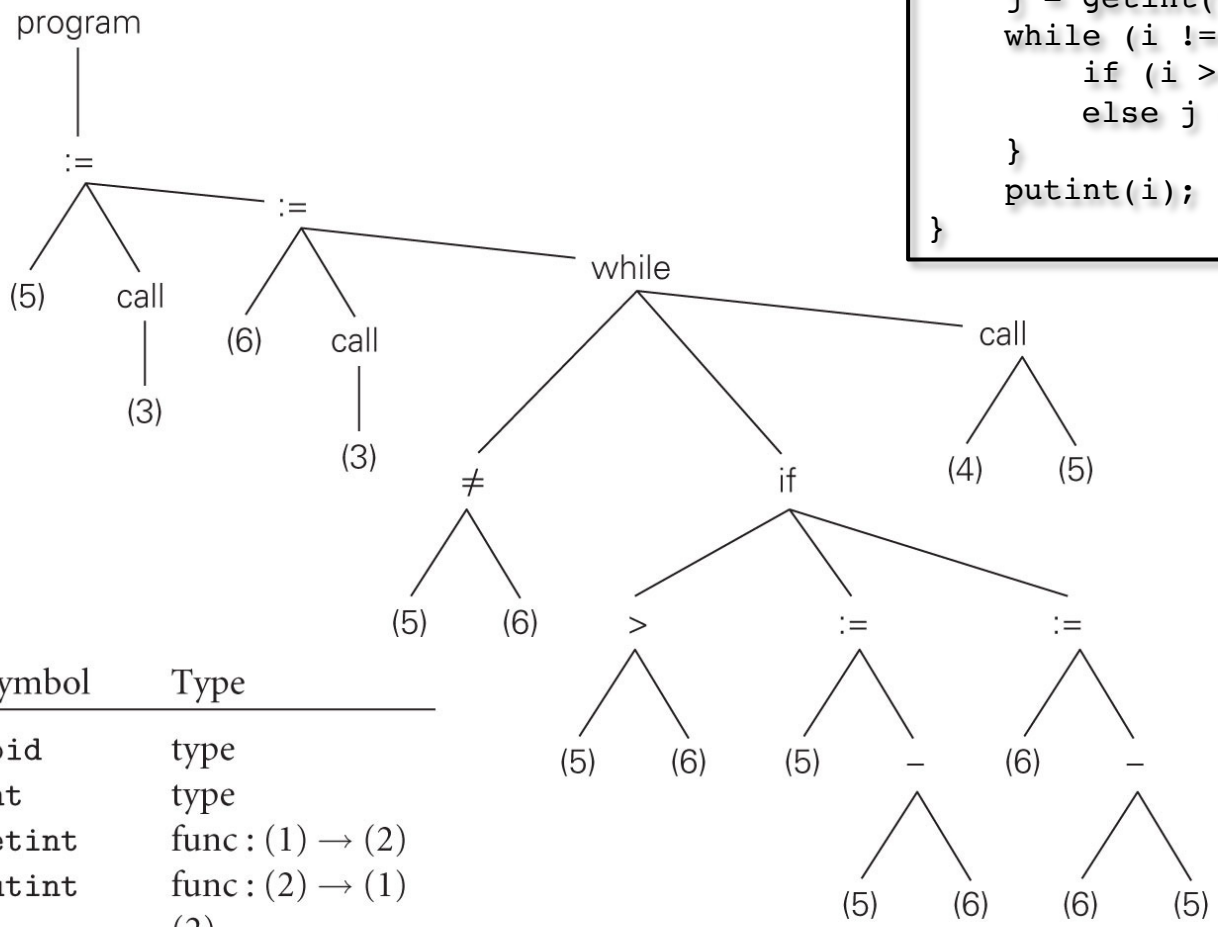
An Overview of Compilation

- Syntax Tree
  - Parse tree = *concrete syntax tree*
    - it shows how the tokens are derived from CFG
    - after that, much information in the parse tree is not relevant
  - Semantic analyzer: parse tree changed into *syntax tree*
    - syntax tree = *abstract syntax tree*
    - removes the “useless” internal nodes
    - annotates the remaining nodes with *attributes*

An Overview of Compilation

- Syntax Tree for GCD program

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



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)

An Overview of Compilation

- Code generation
  - Interpreters use annotated syntax tree to run the program
    - execution means tree traversal
  - Compilers pass the annotated syntax tree as intermediate form to the back end
- Target code generation
  - produces assembly language
  - Example for GCD program – next slide
    - naïve code
    - good code is difficult to produce
    - That's why you'll always find good jobs!

An Overview of Compilation

```
    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
```

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

An Overview of Compilation

- Optimization (code improvement)
  - takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - The code on the previous slide becomes:

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