

Introduction to Programming Languages

CSE 307 – Principles of Programming Languages

Stony Brook University

<http://www.cs.stonybrook.edu/~cse307>

Introduction

- At the beginning there was only machine language: a sequence of bits that directly controls a processor, causing it to add, compare, move data from one place to another
- Example: GCD program in x86 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
```

Introduction

- Assembly languages were invented to allow the machine operations to be expressed with mnemonic abbreviations
- Example: GCD program in x86 assembler:

```
    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
```

Introduction

- Assemblers were eventually augmented with elaborate “macro expansion” facilities to permit programmers to define parameterized abbreviations for common sequences of instructions
- Problem: each different kind of computer had to be programmed in its own assembly language
 - People began to wish for a machine-independent languages
- These wishes led in the mid-1950s to the development of standard higher-level languages compiled for different architectures by *compilers*

Introduction

- Today there are thousands of high-level programming languages, and new ones continue to emerge
- Why are there so many?
 - Evolution
 - Special Purposes
 - Personal Preference

Introduction

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

Introduction

- Why do we have programming languages? 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 implementor's point of view

Why study programming languages?

- Help you choose a language:
 - C vs. C++ for systems programming
 - Matlab vs. Python vs. R for numerical computations
 - Android vs. Java vs. ObjectiveC vs. Javascript for embedded systems
 - Python vs. Ruby vs. Common Lisp vs. Scheme vs. ML for symbolic data manipulation
 - Java RPC (JAX-RPC) vs. C/CORBA for networked PC programs

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.
 - Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European)

Why study programming languages?

- Help you make better use of whatever language you use
 - understand obscure features:
 - In C, help you understand unions, arrays & pointers, separate compilation, catch and throw
 - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals

Why study programming languages?

- Help you make better use of whatever language you use
- understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
 - use simple arithmetic ops. (use $x*x$ instead of $x**2$)
 - 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 Fibonacci function)
 - use C pointers or Pascal "with" statement to factor address calculations

Why study programming languages?

- Help you make better use of whatever language you use
- figure out how to do things in languages that don't support them explicitly:
 - 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)
 - lack of suitable control structures in Fortran
 - use comments and programmer discipline for control structures
 - 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

Classifications

- Many classifications group languages as:
 - imperative
 - von Neumann (Fortran, Pascal, Basic, C)
 - object-oriented (Smalltalk, Eiffel, C++?)
 - scripting languages (Perl, Python, JavaScript, PHP)
 - declarative
 - functional (Scheme, ML, pure Lisp, FP)
 - logic, constraint-based (Prolog, VisiCalc, RPG)
- Many more classifications: markup languages, assembly languages, query languages for DBs, controlled natural languages, etc.

HW1 (part of hw1)

- Write and test the GCD Program in different languages, like C, Prolog, SML and Python:

- In C:

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

- In XSB Prolog:

```
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) :- A < B, C is B-A, gcd(C,A,G).
```

- In SML:

```
fun gcd(m,n):int = if m=n then n
= else if m>n then gcd(m-n,n)
= else gcd(m,n-m);
```

Due: on Blackboard.

- In Python:

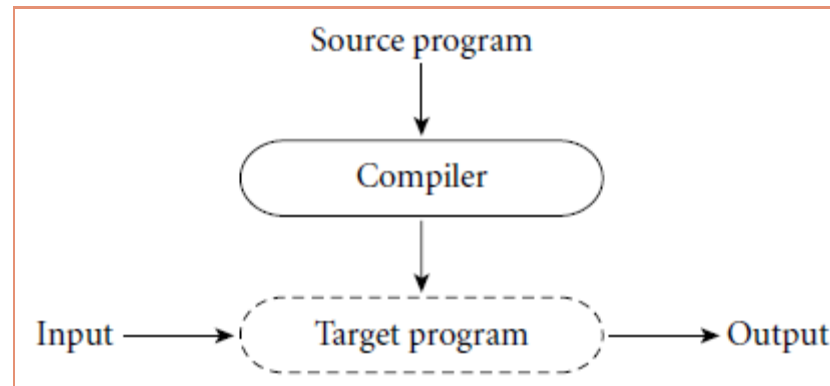
```
def gcd(a, b):
    if a == b:
        return a
    else:
        if a > b:
            return gcd(a-b, b)
        else:
            return gcd(a, b-a)
```

Imperative languages

- Imperative languages, particularly the von Neumann languages, predominate in industry
 - data is modified in memory
 - both control flow and data flow interleaved
 - harder to analyze and optimize than other paradigms (like functional and logic programming)

Compilation vs. Interpretation

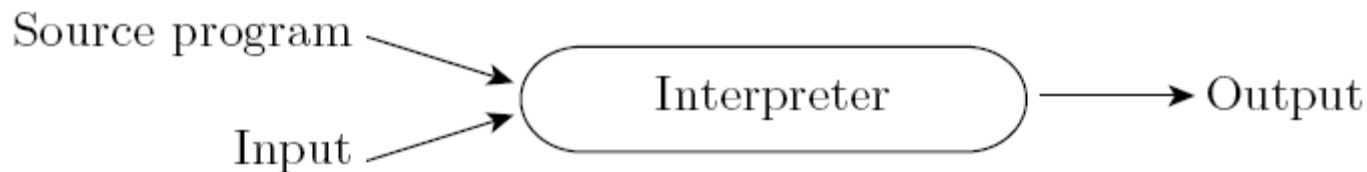
- Compilation vs. interpretation
 - not opposites
 - not a clear-cut distinction
- ***Pure Compilation:***
 - The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



(c) Paul Fodor (CS Stony Brook) and Elsevier

Compilation vs. Interpretation

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

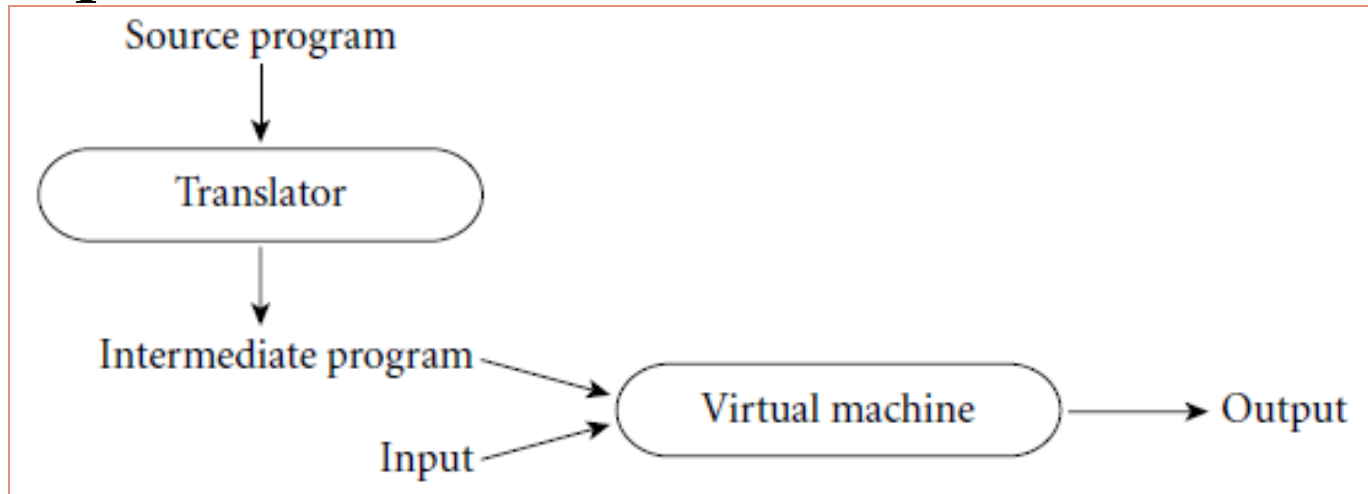


Compilation vs. Interpretation

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

Compilation vs. Interpretation

- Common case is compilation or simple pre-processing, followed by interpretation
- Most modern language implementations include a **mixture** of both compilation and interpretation (Java, C#, ObjectiveC, Dart)



Compilation vs. Interpretation

- 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.

Compilation vs. Interpretation

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

Compilation vs. Interpretation

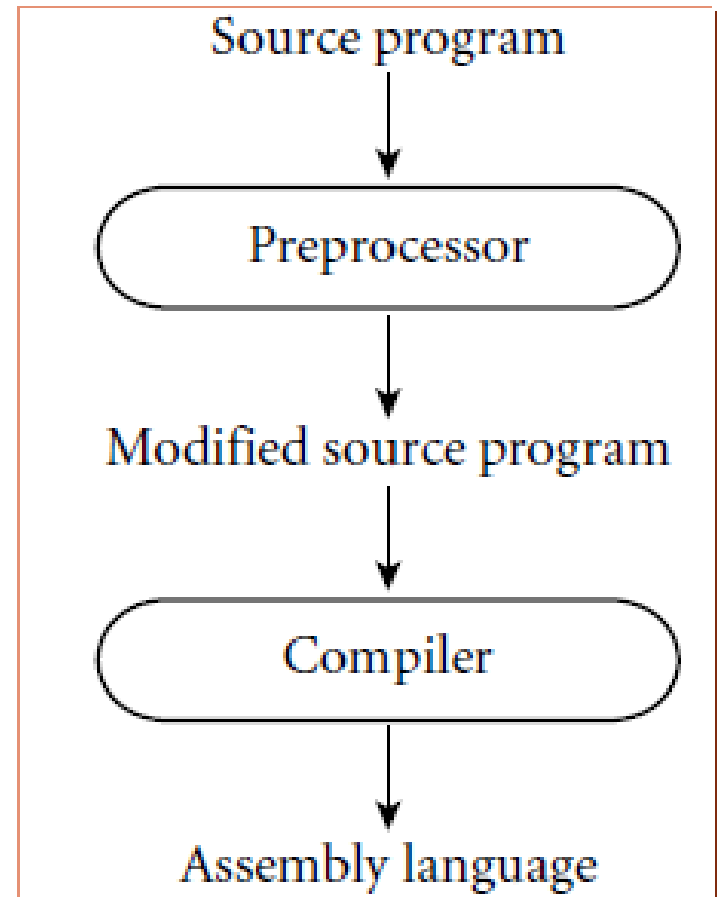
- Implementation strategies:
 - Preprocessor
 - Removes comments and white space
 - Groups characters into tokens (keywords, identifiers, numbers, symbols)
 - Expands abbreviations in the style of a macro assembler
 - Identifies higher-level syntactic structures (loops, subroutines)

Compilation vs. Interpretation

- Implementation strategies:

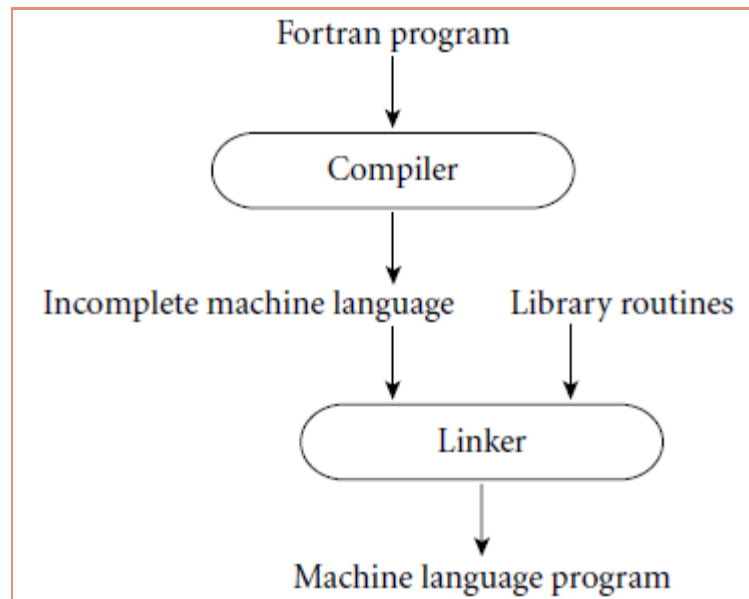
- The C Preprocessor:

- removes comments
- expands macros



Compilation vs. Interpretation

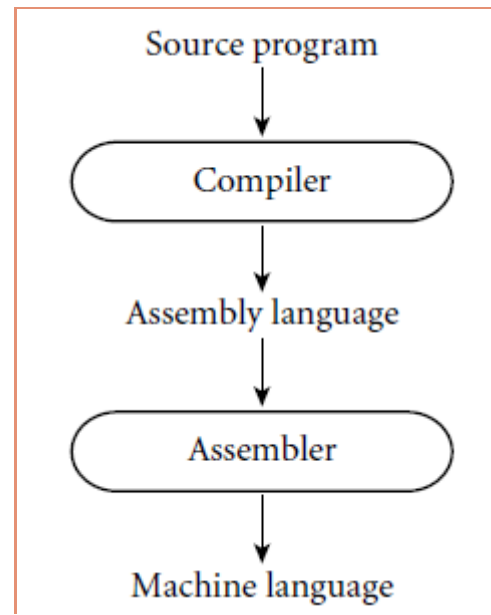
- Implementation strategies:
 - Library of Routines and *Linking*
 - Compiler uses a linker program to merge the appropriate library of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:



(c) Paul Fodor (CS Stony Brook) and Elsevier

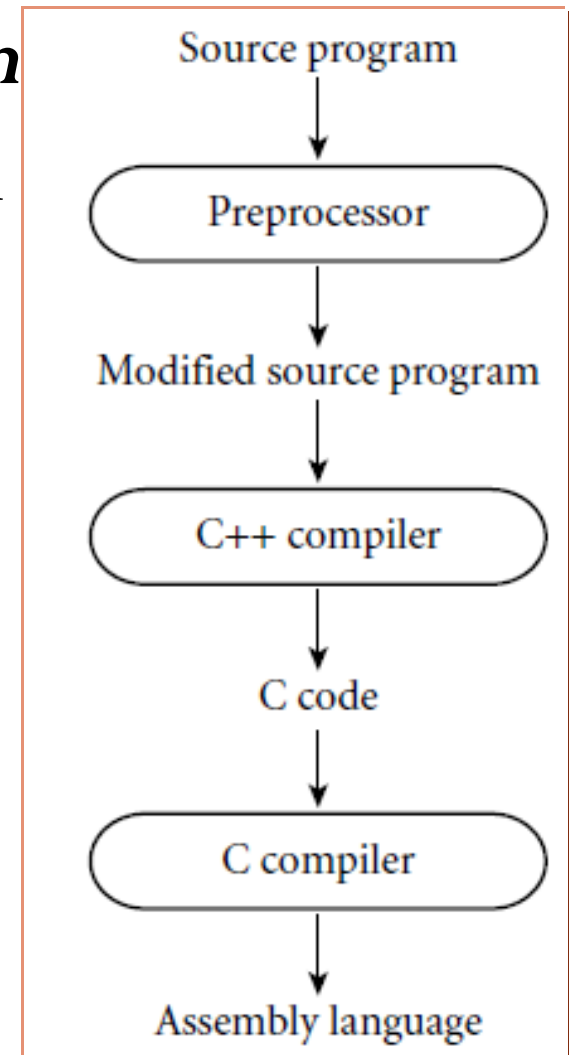
Compilation vs. Interpretation

- Implementation strategies:
 - 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)



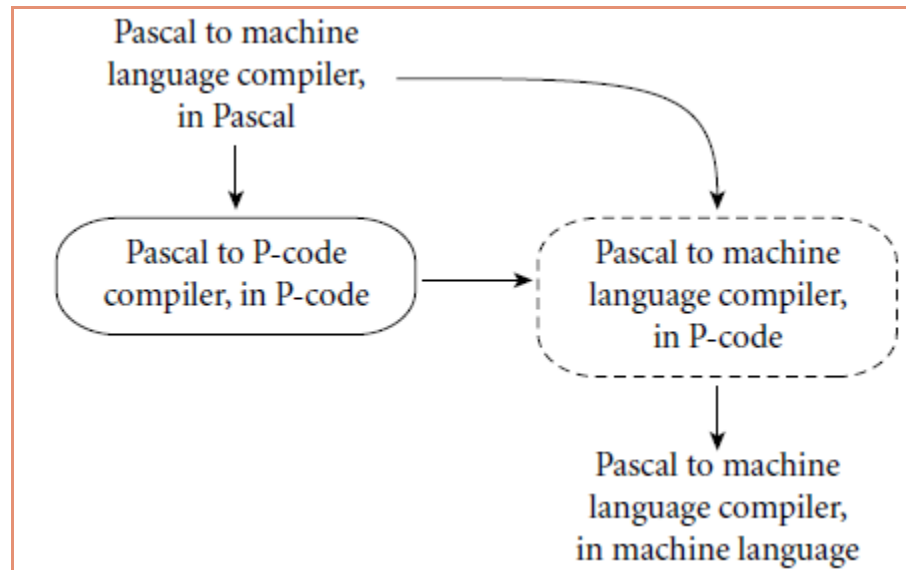
Compilation vs. Interpretation

- Implementation strategies:
 - ***Source-to-Source Translation***
 - C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language



Compilation vs. Interpretation

- Implementation strategies:
 - **Bootstrapping**: many compilers are self-hosting: they are written in the language they compile
 - How does one compile the compiler in the first place?
 - Response: one starts with a simple implementation—often an interpreter—and uses it to build progressively more sophisticated versions



Compilation vs. Interpretation

- Implementation strategies:
 - Compilation of Interpreted Languages (e.g., Prolog, Lisp, Smalltalk, Java 9 jlink, C#):
 - 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.
 - Permit a lot of late binding

Compilation vs. Interpretation

- Implementation strategies:
 - *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** (e.g., dynamic indexing in Prolog).
 - The Java language definition defines a machine-independent intermediate form known as byte code. Bytecode is the standard format for distribution of Java programs:
 - it allows programs to be transferred easily over the Internet, and then run on any platform
 - The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

Compilation vs. Interpretation

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

Compilation vs. Interpretation

- 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 is still necessary
 - E.g., XSB Prolog is compiled into .wam (Warren Abstract Machine) files and then executed by the interpreter
- **Unconventional compilers:**
 - text formatters: TEX and troff are actually compilers
 - silicon compilers: laser printers themselves incorporate interpreters for the Postscript page description language
 - query language processors for database systems are also compilers: translate languages like SQL into primitive operations (e.g., tuple relational calculus and domain relational calculus)

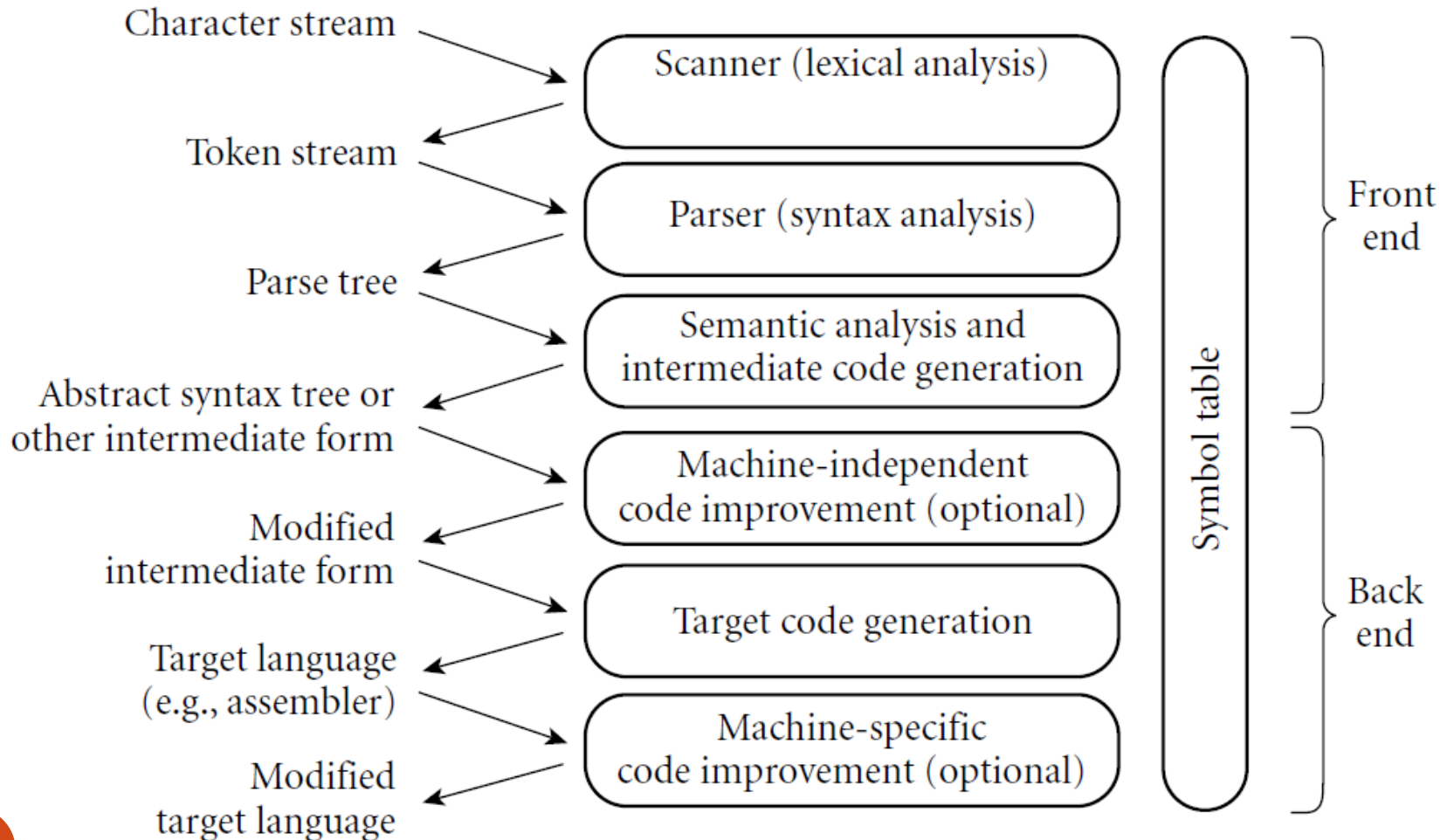
Programming Environment Tools

- Tools/IDEs:
 - Compilers and interpreters do not exist in isolation
 - Programmers are assisted by tools and IDEs

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

An Overview of Compilation

• Phases of Compilation



An Overview of Compilation

- 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 (Deterministic finite automaton)

An Overview of Compilation

- *Parsing* is recognition of a context-free language, e.g., via PDA (Pushdown automaton)
- Parsing discovers the "context free" structure of the program
- Informally, it finds the structure you can describe with syntax diagrams (e.g., the "circles and arrows" in a language manual)

An Overview of Compilation

- *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.

An Overview of Compilation

- *Intermediate Form* (IF) is 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

An Overview of Compilation

- 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

An Overview of Compilation

- *Code generation* phase produces assembly language or (sometime) relocatable machine language
- Certain machine-specific optimizations (use of special instructions or addressing modes, etc.) may be performed during or after target code generation

An Overview of Compilation

- **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

An Overview of Compilation

- Example, take the 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);
}
```

An Overview of Compilation

- 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    )    ;  
}
```

An Overview of Compilation

- 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

An Overview of Compilation

- Context-Free Grammar and Parsing

- Grammar Example for while loops in C:

while-iteration-statement \rightarrow *while (expression) statement*
statement, in turn, is often a list enclosed in braces:

statement \rightarrow *compound-statement*

compound-statement \rightarrow { *block-item-list opt* }

where

block-item-list opt \rightarrow *block-item-list*

or

block-item-list opt \rightarrow ϵ

and

block-item-list \rightarrow *block-item*

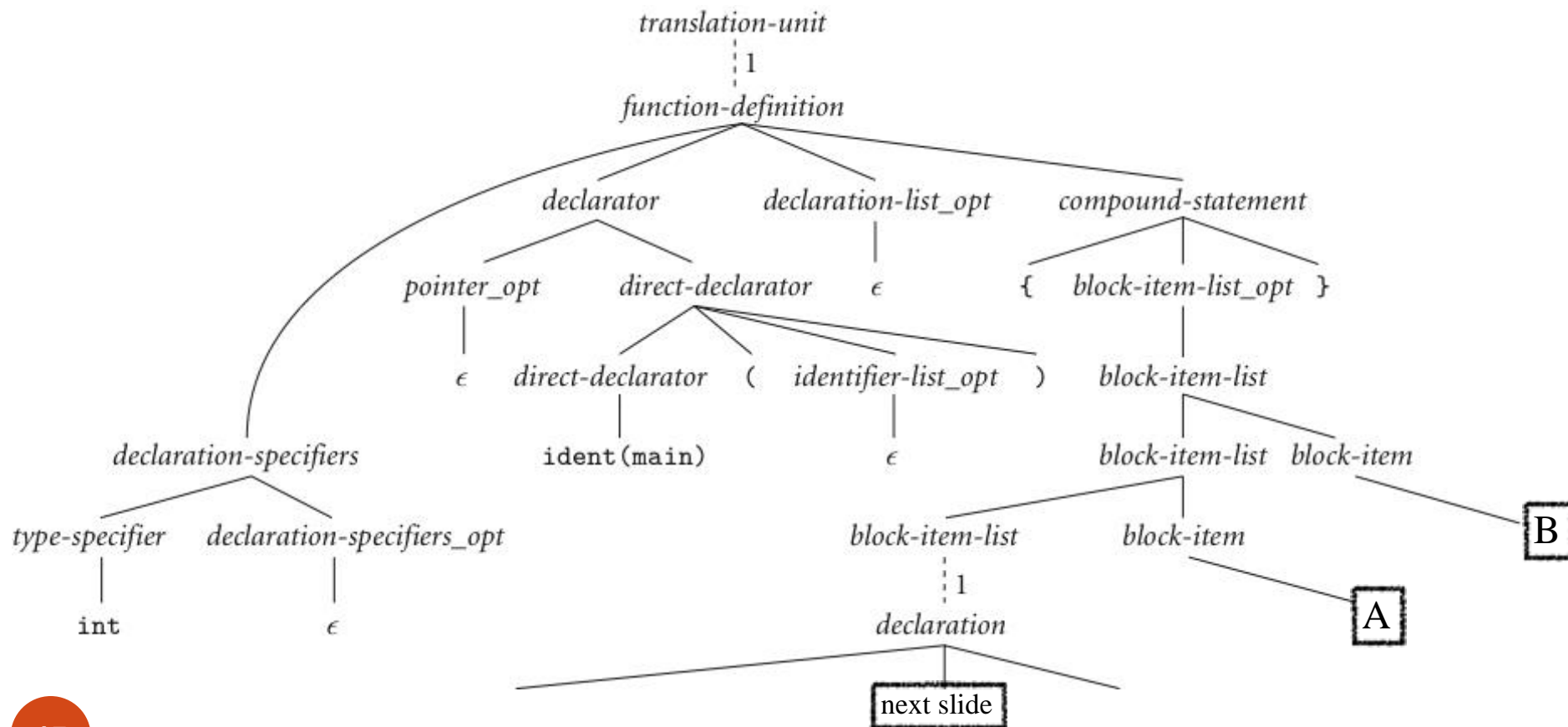
block-item-list \rightarrow *block-item-list block-item*

block-item \rightarrow *declaration*

block-item \rightarrow *statement*

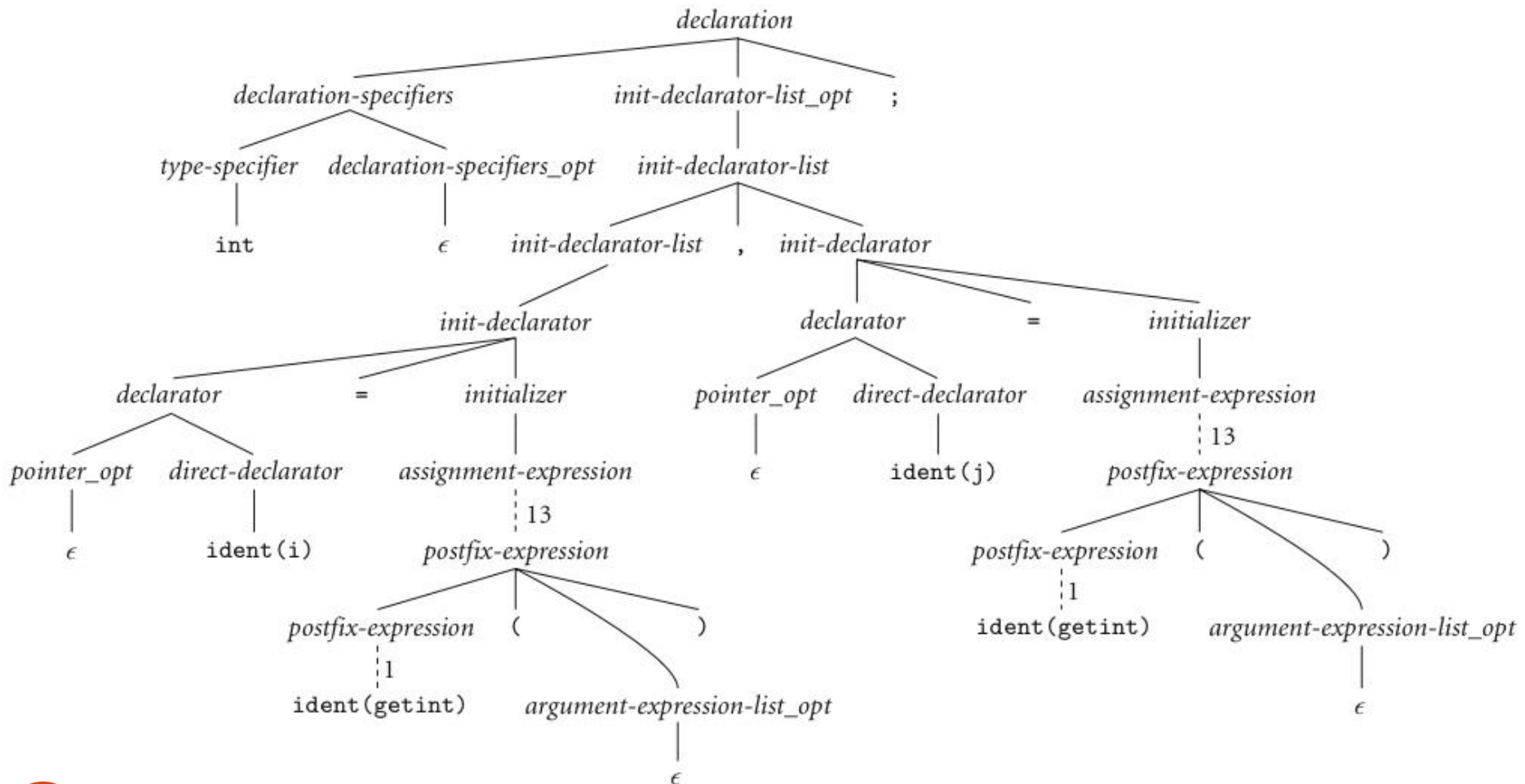
An Overview of Compilation

- Context-Free Grammar and Parsing
 - GCD Program **Parse Tree**:



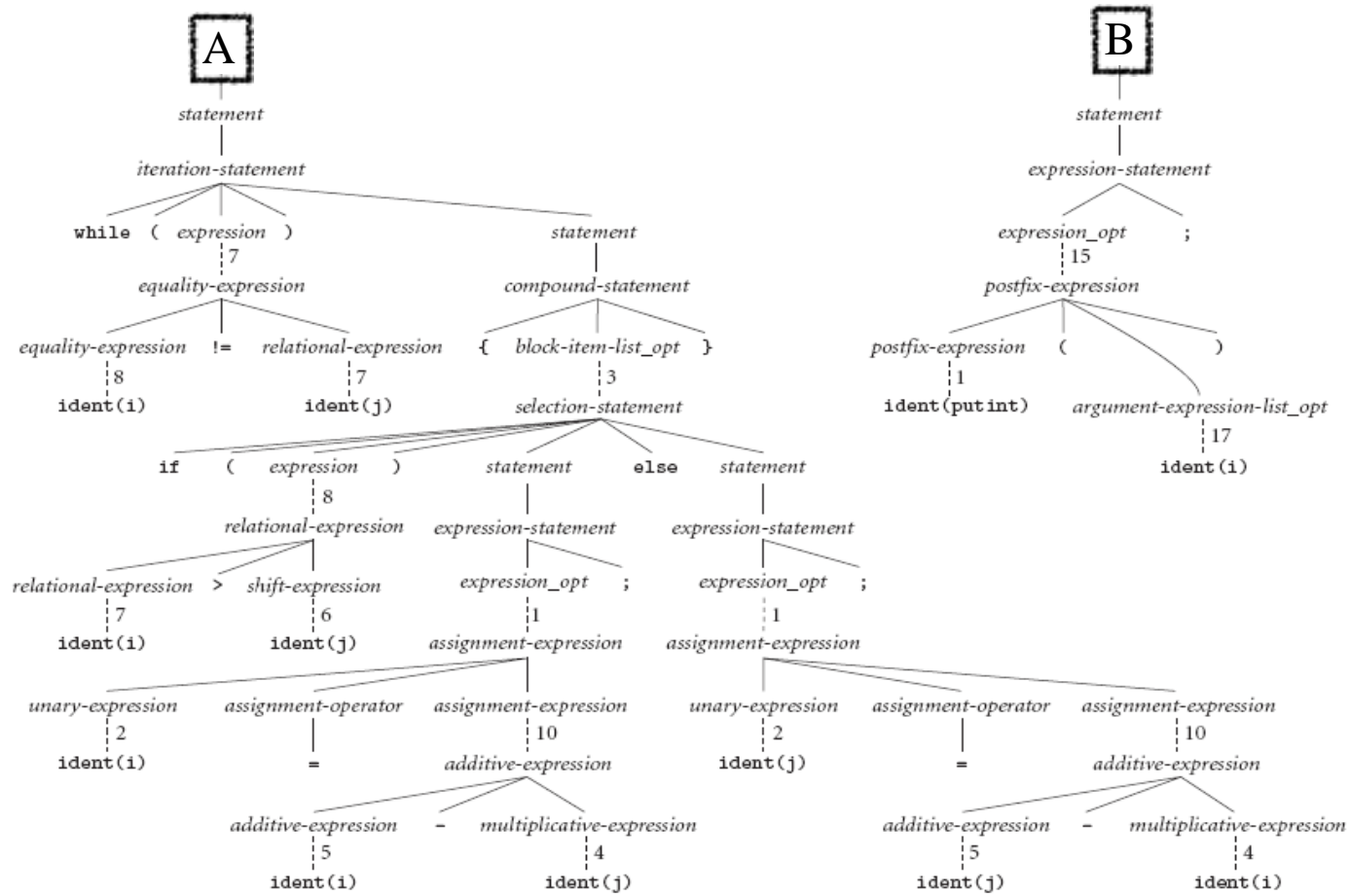
An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



An Overview of Compilation

- **Semantic Analysis and Intermediate Code Generation**
- Semantic analysis is the discovery of meaning in a program
 - tracks the types of both identifiers and expressions
 - builds and maintains a *symbol table* data structure that maps each identifier to the information known about it
 - context checking
 - Every identifier is declared before it is used
 - No identifier is used in an inappropriate context (e.g., adding a string to an integer)
 - Subroutine calls provide the correct number and types of arguments.
 - Labels on the arms of a switch statement are distinct constants.
 - Any function with a non-void return type returns a value explicitly

An Overview of Compilation

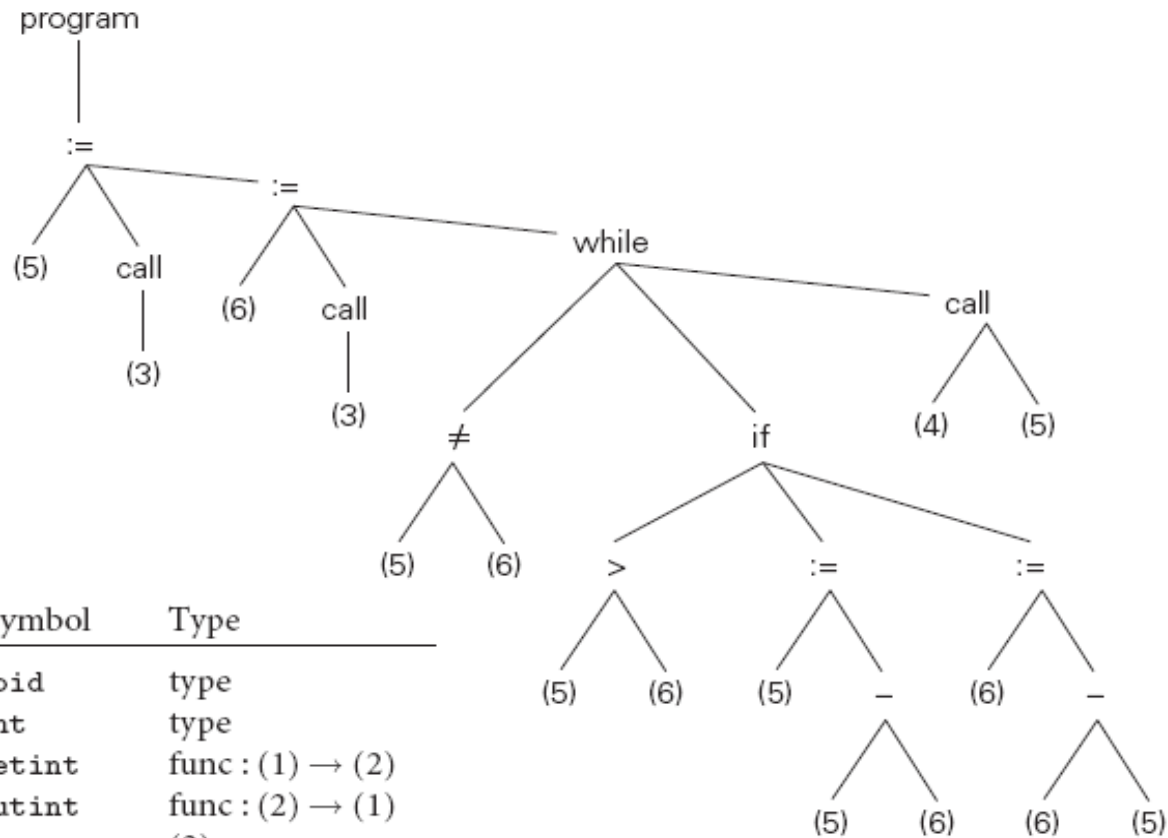
- Semantic analysis implementation
 - *semantic action routines* are invoked by the parser when it realizes that it has reached a particular point within a grammar rule.
- Not all semantic rules can be checked at compile time: only the *static semantics* of the language
 - the *dynamic semantics* of the language must be checked at run time
 - Array subscript expressions lie within the bounds of the array
 - Arithmetic operations do not overflow

An Overview of Compilation

- Semantic Analysis and Intermediate Code Generation
 - **The parse tree is very verbose:** once we know that a token sequence is valid, **much of the information in the parse tree is irrelevant to further phases of compilation**
 - The semantic analyzer typically transforms the parse tree into an *abstract syntax tree* (*AST* or simply a *syntax tree*) by removing most of the “artificial” nodes in the tree’s interior
 - The semantic analyzer also *annotates* the remaining nodes with useful information, such as pointers from identifiers to their symbol table entries
 - The annotations attached to a particular node are known as its *attributes*

An Overview of Compilation

- GCD Syntax Tree (AST)



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

- In many compilers, the annotated syntax tree constitutes the *intermediate form* that is passed from the front end to the back end.
- In other compilers, semantic analysis ends with a **traversal** of the tree that generates some other intermediate form
 - One common such form consists of a control flow graph whose nodes resemble fragments of assembly language for a simple idealized machine

An Overview of Compilation

- **Target Code Generation:**
 - The code generation phase of a compiler translates the intermediate form into the target language
 - To generate assembly or machine language, the code generator traverses the symbol table to assign locations to variables, and then traverses the intermediate representation of the program, generating loads and stores for variable references, interspersed with appropriate arithmetic operations, tests, and branches

An Overview of Compilation

- Target Code Generation:
 - Naive x86 assembly language for the GCD program

```
    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
```

An Overview of Compilation

- Some improvements are machine independent
- Other improvements require an understanding of the target machine
- Code improvement often appears as two phases of compilation, one immediately after semantic analysis and intermediate code generation, the other immediately after target code generation