# Introduction to Programming Languages

CSE 307 – Principles of Programming Languages

SUNY Korea

# Introduction

- Computer users usually don't think about the billions of tiny electronic operations that go on each second.
- The situation is (very roughly) similar to when you are driving your car. You think about the "big operations" it can perform, such as "accelerate", "turn left", "brake", and so on.
- You don't think about tiny operations, such as the valves in your engine opening and closing 24,000 times per minute or the crankshaft spinning at 3000 revolutions per minute.
- 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.

# Machine Instructions

- A machine instruction consists of several bytes in memory that tell the processor to perform one machine operation.
- The processor looks at machine instructions in main memory one after another, and performs one machine operation for each machine instruction.
- The collection of machine instructions in main memory is called a machine language program or (more commonly) an executable program.
- Actual processors have many more machine instructions and the instructions are much more detailed. A typical processor has a thousand or more different machine instructions.
- https://chortle.ccsu.edu/java5/Notes/chap04/ch04_4.html

# GCD Program in x86

```
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
```

- This program calculates GCD (Greatest Common Divider) of two integers using Euclid's algorithm.
- Written in machine language expressed as hexadecimal (base 16) numbers.
- Instruction set used is x86.
- It can be seen that writing larger programs quickly becomes error-prone.

# Assembly Languages

- Assembly languages were invented to allow operations to be expressed with mnemonic abbreviations.
- The low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.
- Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

# Assembly Language Syntax

```
INC COUNT        ; Increment the memory variable COUNT
MOV TOTAL, 48    ; Transfer the value 48 in the
                 ; memory variable TOTAL
ADD AH, BH       ; Add the content of the
                 ; BH register into the AH register
AND MASK1, 128   ; Perform AND operation on the
                 ; variable MASK1 and 128
ADD MARKS, 10    ; Add 10 to the variable MARKS
MOV AL, 10       ; Transfer the value 10 to the AL register
```

# Hello World in Assembly Language

```
section .text
   global _start     ;must be declared for linker (ld)

_start:              ;tells linker entry point
   mov   edx,len     ;message length
   mov   ecx,msg     ;message to write
   mov   ebx,1       ;file descriptor (stdout)
   mov   eax,4       ;system call number (sys_write)
   int   0x80        ;call kernel

   mov   eax,1       ;system call number (sys_exit)
   int   0x80        ;call kernel

section .data
msg db 'Hello, world!', 0xa  ;string to be printed
len equ $ - msg     ;length of the string
```

# GCD program in Assembly Language

```
        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* which translate high-level language code to assembly or machine level language.

# Introduction

- Compilers are more complicated than assemblers.
- One-to-one correspondence between source and target languages does not exist with high-level languages.
- Initial compilers (such as Fortran compilers) were slow as human programmers could also translate code with some efforts.
- Over the time, performance gap narrowed and eventually reversed.
- Better hardware and improvements in compiler technology generate code better and faster compared to a human being.

# Introduction

- Today there are thousands of high-level programming languages, and new ones continue to emerge. Why are there so many?
  - Evolution
    - E.g. goto-based control flow to while loop, case-switch statements
    - Object orientation (C++, Java), rapid development (python)
  - Special Purposes
    - Awk for string manipulation, C is good for low level system programming
  - Personal Preference
    - Terseness of C (using few words), recursive vs. iteration, pointers vs. not using pointers

# Introduction

- What makes a language successful?
  - easy to learn (python, BASIC, Pascal, LOGO, Scheme)
  - easy to express things (abstraction), easy use once fluent, "powerful" (C, Java, Common Lisp, APL, Algol-68, Perl)
  - easy to implement (Javascript, BASIC, Forth)
  - Easily available (portable copies of Pascal sent to universities)
  - possible to compile to very good (fast/small) code (Fortran, C)
  - Open source compiler or interpreter

# Introduction

- What makes a language successful?
  - Standardization of language and libraries to ensure effective portability of code across platforms (C vs. Java)
  - backing of a powerful sponsor (Java – SUN/Oracle, Visual Basic, COBOL , Ada – US Defense, PL/1 - IBM)
  - wide dissemination at minimal cost (Java, Pascal, Turing, erlang)
  - Choosing optimal language is a tradeoff
  - Consider viewpoints of programmer and implementor
  - Cost of implementation

# Introduction

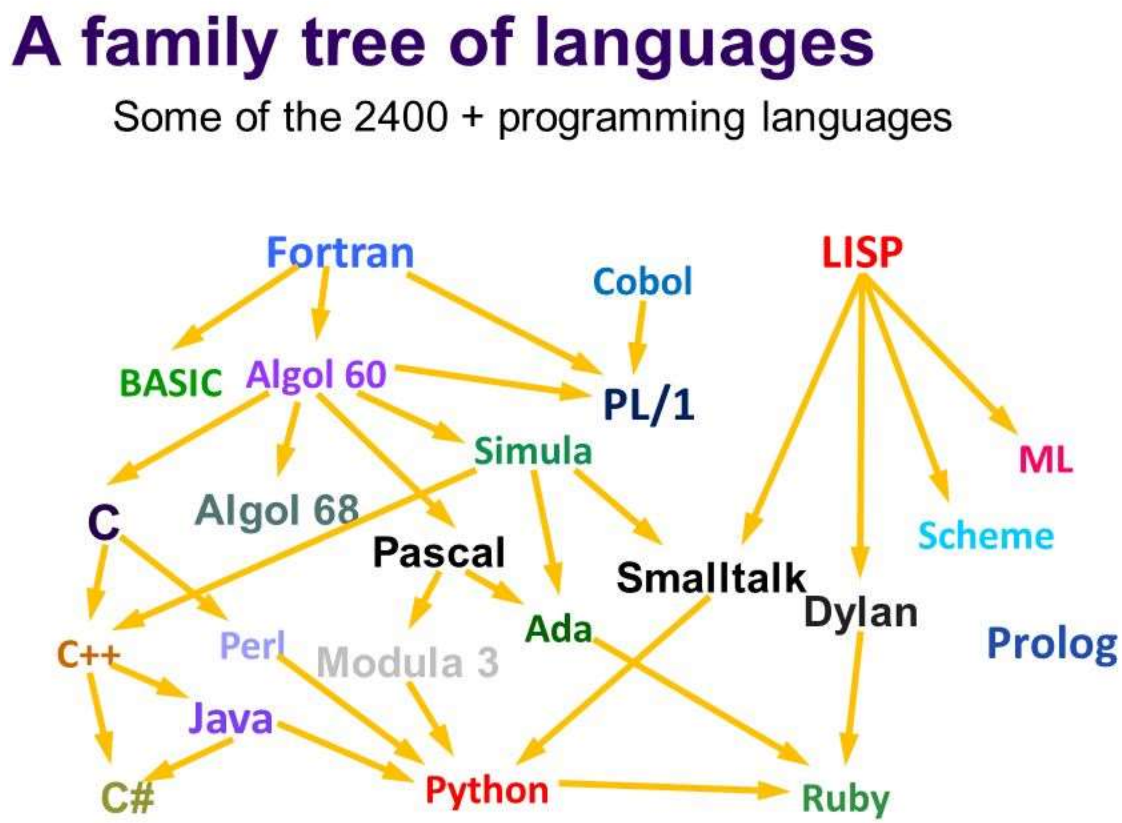
- Why do we have programming languages? What is a language for?
  - way of thinking -- way of expressing algorithms
  - languages from the user's point of view
  - abstraction of virtual machine -- way of specifying what you want
  - the hardware to do without getting down into the bits
  - languages 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 (not purely numerical) 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



# Why study programming languages?

- 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 – Albanian, Armenian, Balto-Slavic, Baltic, Slavic, Celtic, Germanic).

# 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

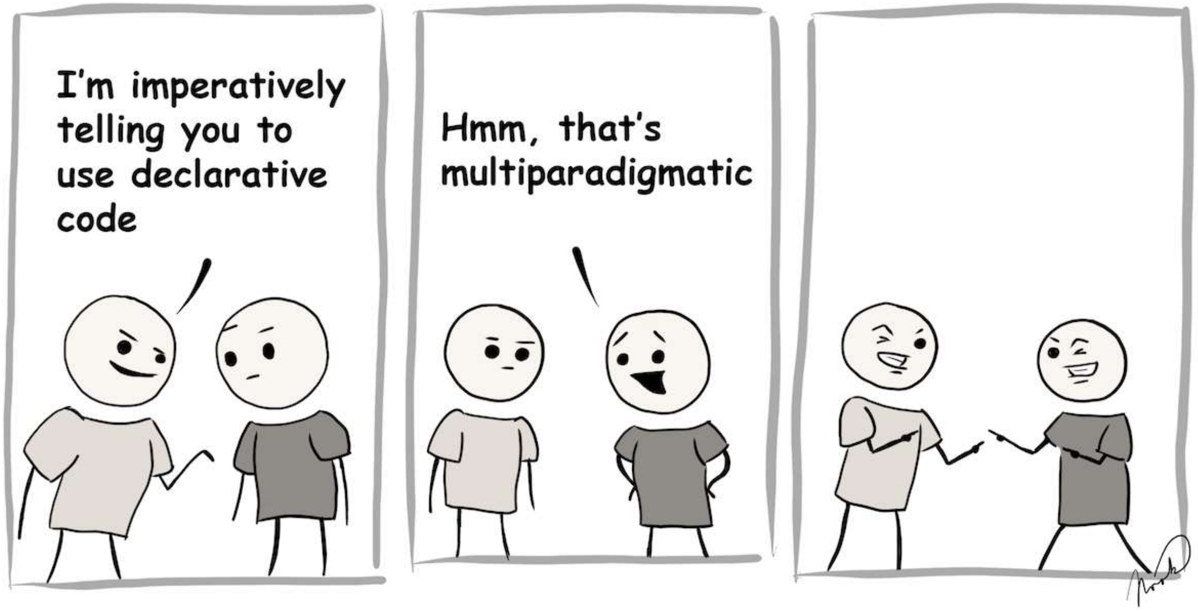
# Why study programming languages?

- understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
  - use simple arithmetic equal (use x*x instead of x**2)
  - Avoid unnecessary temporary variables and use copy constructors to minimize the cost of initialization

```
#include<iostream>
using namespace std;

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here
```

# Why study programming languages?

- 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




- **Imperative programming:** Telling the "machine" how to do something, and as a result what you want to happen will happen. (e.g. Java code)
- **Declarative programming:** Telling the "machine"1 what you would like to happen, and let the computer figure out how to do it. (e.g. HTML code, functional programming code)

# Imperative Example

```
function double (arr) {
  let results = []
  for (let i = 0; i < arr.length; i++){
    results.push(arr[i] * 2)
  }
  return results
}
```

```
function add (arr) {
  let result = 0
  for (let i = 0; i < arr.length; i++){
    result += arr[i]
  }
  return result
}
```

# Declarative Example
# SQL, HTML

```sql
SELECT * FROM Users WHERE Country='Mexico';
```

```html
<article>
  <header>
    <h1>Declarative Programming</h1>
    <p>Sprinkle Declarative in your verbiage to sound smart</p>
  </header>
</article>
```

# Classifications

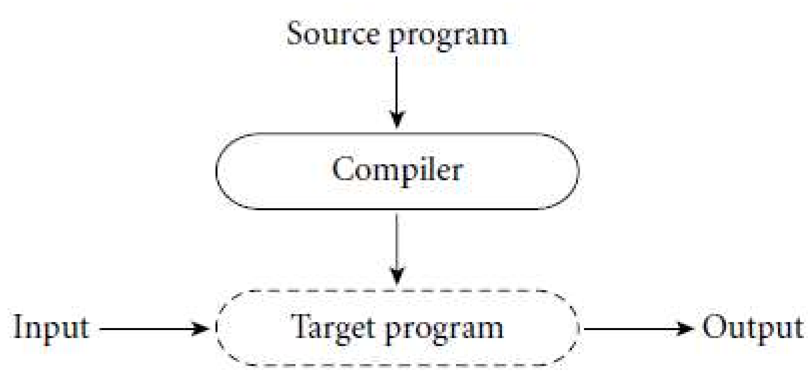
- Many classifications group languages as:
  - imperative
    - von Neumann (Fortran, Pascal, Basic, C)
    - object-oriented (Smalltalk, Eiffel, C++?)
    - scripting languages (Perl, Python, JavaScript, PHP)
  - declarative
    - functional (Scheme, ML, pure Lisp, FP)
    - logic, constraint-based (Prolog, VisiCalc, RPG)
    - Markup languages (HTML)

# Imperative languages

- Imperative languages, particularly the von Neumann languages, predominate in industry

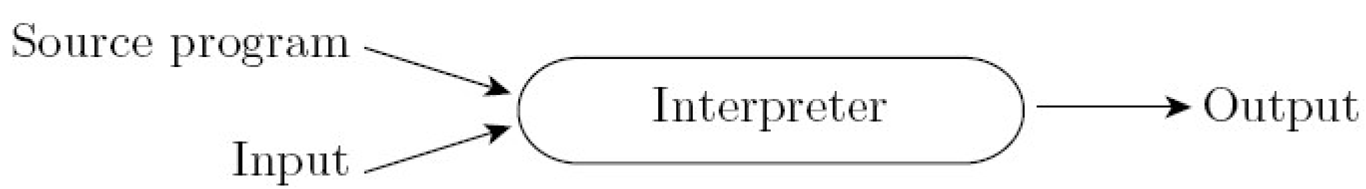
# Compilation vs. Interpretation

- 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:



# Compilation vs. Interpretation

- Pure Interpretation
  - Interpreter stays around for the execution of the program
  - Interpreter is the locus of control during execution
  - Some language features are impossible without interpreter – e.g. in Lisp, program write new pieces of itself and execute on the fly

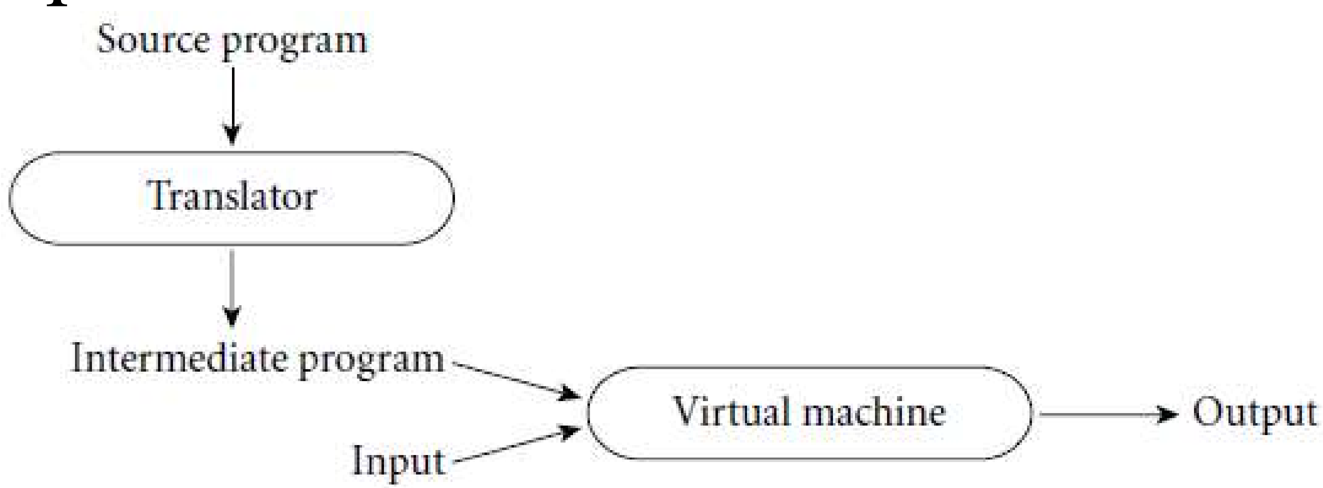


# Compilation vs. Interpretation

- Interpretation:
  - Greater flexibility and portability
  - Better diagnostics (debugging and 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



# Compilation vs. Interpretation

- Many compiled languages have interpreted pieces, e.g., formats in Fortran or C
- Some compilers produce nothing but virtual instructions, e.g., Java bytecode, Pascal P-code, Microsoft COM+ (.net)

# Compilation vs. Interpretation

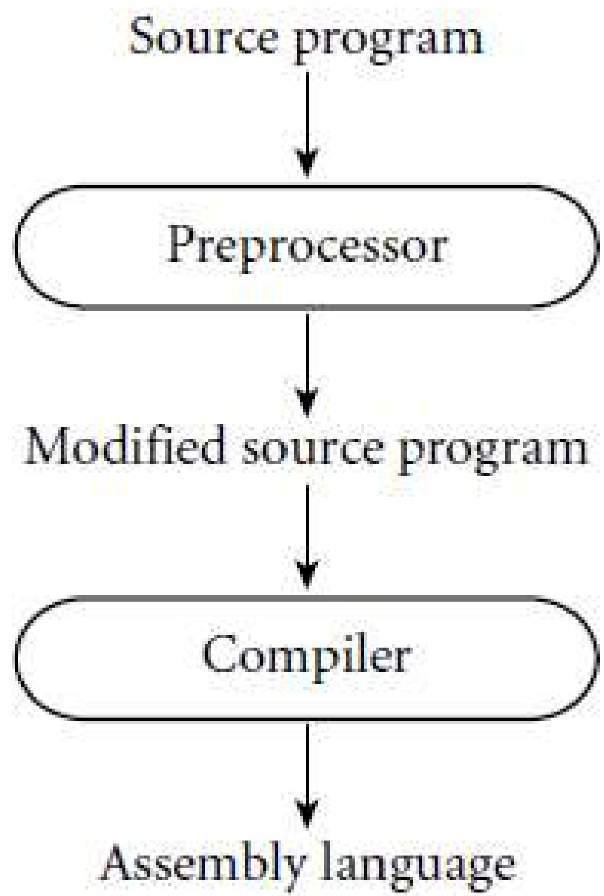
- Implementation strategies:
  - Preprocessor
    - Removes comments and white space
    - Groups characters into tokens (keywords, identifiers, numbers, symbols)
    - Expands abbreviations in the style of a macro assembler
    - Identifies higher-level syntactic structures (loops, subroutines)

# Compilation vs. Interpretation

- 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 may do formatting, remove comments etc. but will often let errors through.

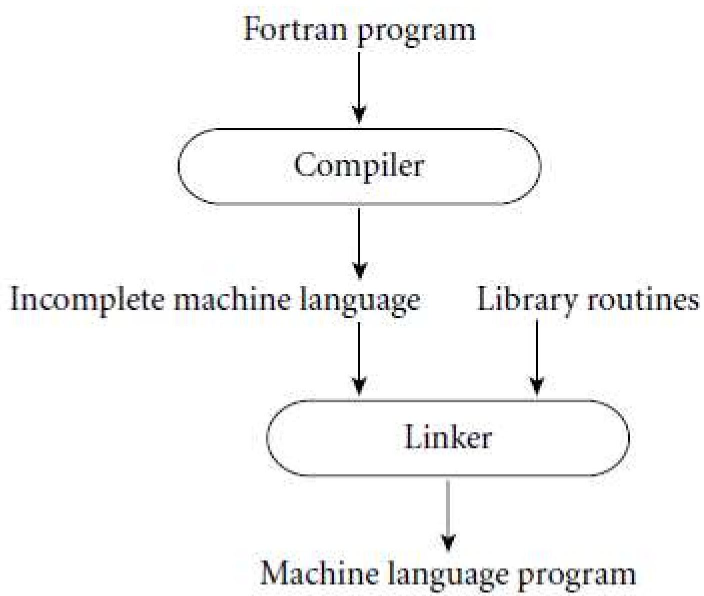
# Compilation vs. Interpretation

- Implementation strategies:
  - The C Preprocessor:
    - removes comments
    - expands macros



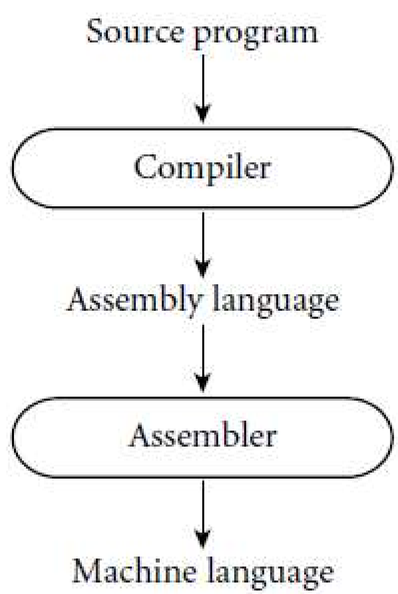
# Compilation vs. Interpretation

- 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:



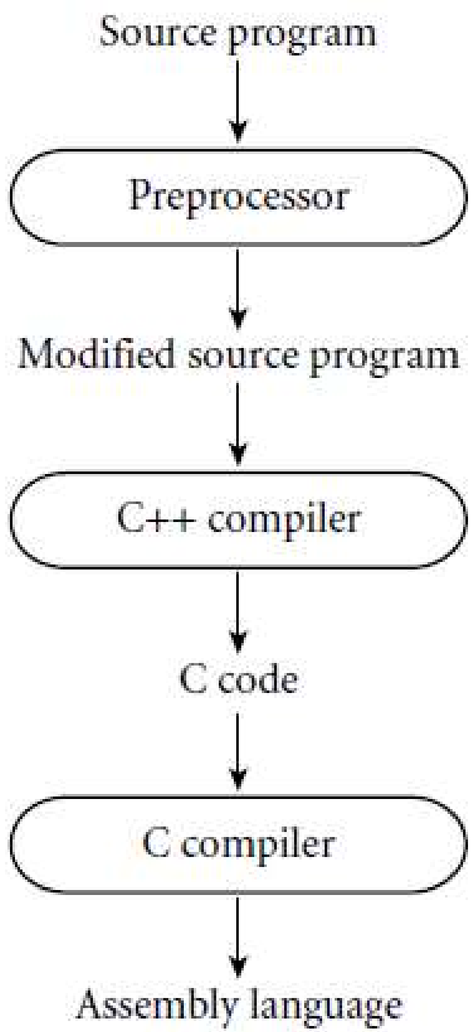
# Compilation vs. Interpretation

- 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

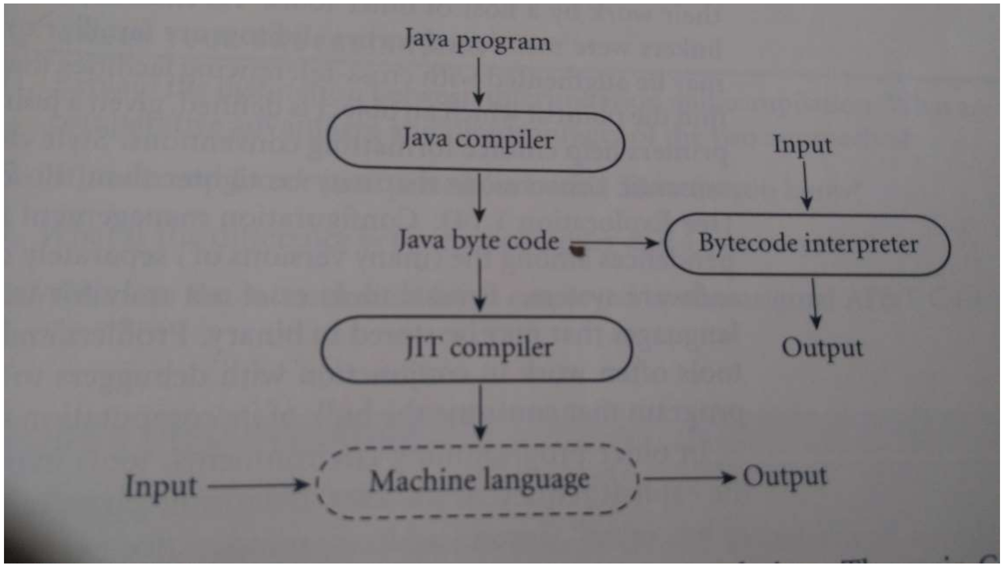
- Implementation strategies:
  - Bootstrapping: many compilers are self-hosting: they are written in the language they compile
    - How does one compile the compiler in the first place?
    - Response: one starts with a simple implementation—often an interpreter—and uses it to build progressively more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

# Compilation vs. Interpretation

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

# 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, Prolog, java, C#)
    - The Java language definition defines a machine-independent intermediate form known as byte code. Bytecode is the standard format for distribution of Java programs that allows programs to be transferred easily over the Internet, and then run on any platform



# 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.
- **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 | Id, Id-so |
| Perusal tools | More, less, od, nm |
| Program cross-reference | ctags |

# `indent`: Indent and Format C Program Source

This is Edition 2.2.10 of *The* `indent` *Manual*, for Indent Version 2.2.10, last updated 23 July 2008.



The `indent` program changes the appearance of a C program by inserting or deleting whitespace.

This is Edition 2.2.10, 23 July 2008, of *The* `indent` *Manual*, for Indent Version 2.2.10.

# cpp(1) - Linux man page

## Name

cpp - The C Preprocessor

## Synopsis

cpp [**-D***macro*[=*defn*]...] [**-U***macro*] [**-I***dir*...] [**-iquote***dir*...] [**-W***warn*...] [**-M**|**-MM**] [**-MG**] [**-MF** *filename*] [**-MP**] [**-MQ** *target*...] [**-MT** *target*...] [**-P**] [**-fno-working-directory**] [**-x** *language*] [**-std=***standard*] *infile outfile*

Only the most useful options are listed here; see below for the remainder.

## Description

The C preprocessor, often known as *cpp*, is a *macro processor* that is used automatically by the C compiler to transform your program before compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

# Ctags

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(November 2011) (Learn how and when to remove this template message)*

**Ctags**

| | |
|---|---|
| **Developer(s)** | Ken Arnold |
| **Repository** | http://BXR.SU/FreeBSD/usr.bin/ctags/ |
| **Type** | Programming tool (Specifically: Code navigation tool) |

**Ctags** is a programming tool that generates an index (or tag) file of names found in source and header files of various programming languages. Depending on the language, functions, variables, class members, macros and so on may be indexed. These tags allow definitions to be quickly and easily located by a text editor or other utility. Alternatively, there is also an output mode that generates a cross reference file, listing information about various names found in a set of language files in human-readable form.

The original **Ctags** was introduced in BSD Unix and was written by Ken Arnold, with Fortran support by Jim Kleckner and Pascal support by Bill Joy.

# Android Debug Bridge (adb)

☆☆☆☆☆

Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

- **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.
- **A daemon (adbd)**, which runs commands on a device. The daemon runs as a background process on each device.
- **A server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

`adb` is included in the Android SDK Platform-Tools package. You can download this package with the SDK Manager, which installs it at `android_sdk/platform-tools/`. Or if you want the standalone Android SDK Platform-Tools package, you can download it here.

# stylelint

A mighty, modern linter that helps you avoid errors and enforce conventions in your styles.

## Features

It's mighty because it:

- has over **160 built-in rules** to catch errors, apply limits and enforce stylistic conventions
- understands the **latest CSS syntax** including custom properties and level 4 selectors
- parses **CSS-like syntaxes** like SCSS, Sass, Less and SugarSS
- extracts **embedded styles** from HTML, markdown and CSS-in-JS object & template literals
- automatically **fixes** some violations (*experimental feature*)
- supports **plugins** so you can create your own rules or make use of plugins written by the community

## Example output

```
visual.css
 2:12  ✖  Unexpected invalid hex color "#4f"                            color-no-invalid-hex
 4:1   ⚠  Expected ".foo.bar" to have a specificity no more than "0,1,0"  selector-max-specificity
 6:13  ✖  Unexpected unit "px" for property "margin"                    declaration-property-unit-blacklist
 7:17  ✖  Expected single space after "," in a single-line function     function-comma-space-after
```

# Make (software)

From Wikipedia, the free encyclopedia

In software development, **Make** is a build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix and Unix-like operating systems.

Besides building programs, Make can be used to manage any project where some files must be updated automatically from others whenever the others change.

# GNU Assembler

From Wikipedia, the free encyclopedia

The **GNU Assembler**, commonly known as **gas** or simply **as**, its executable name, is the assembler used by the GNU Project. It is the default back-end of GCC. It is used to assemble the GNU operating system and the Linux kernel, and various other software. It is a part of the GNU Binutils package. It was announced in 1986[1].

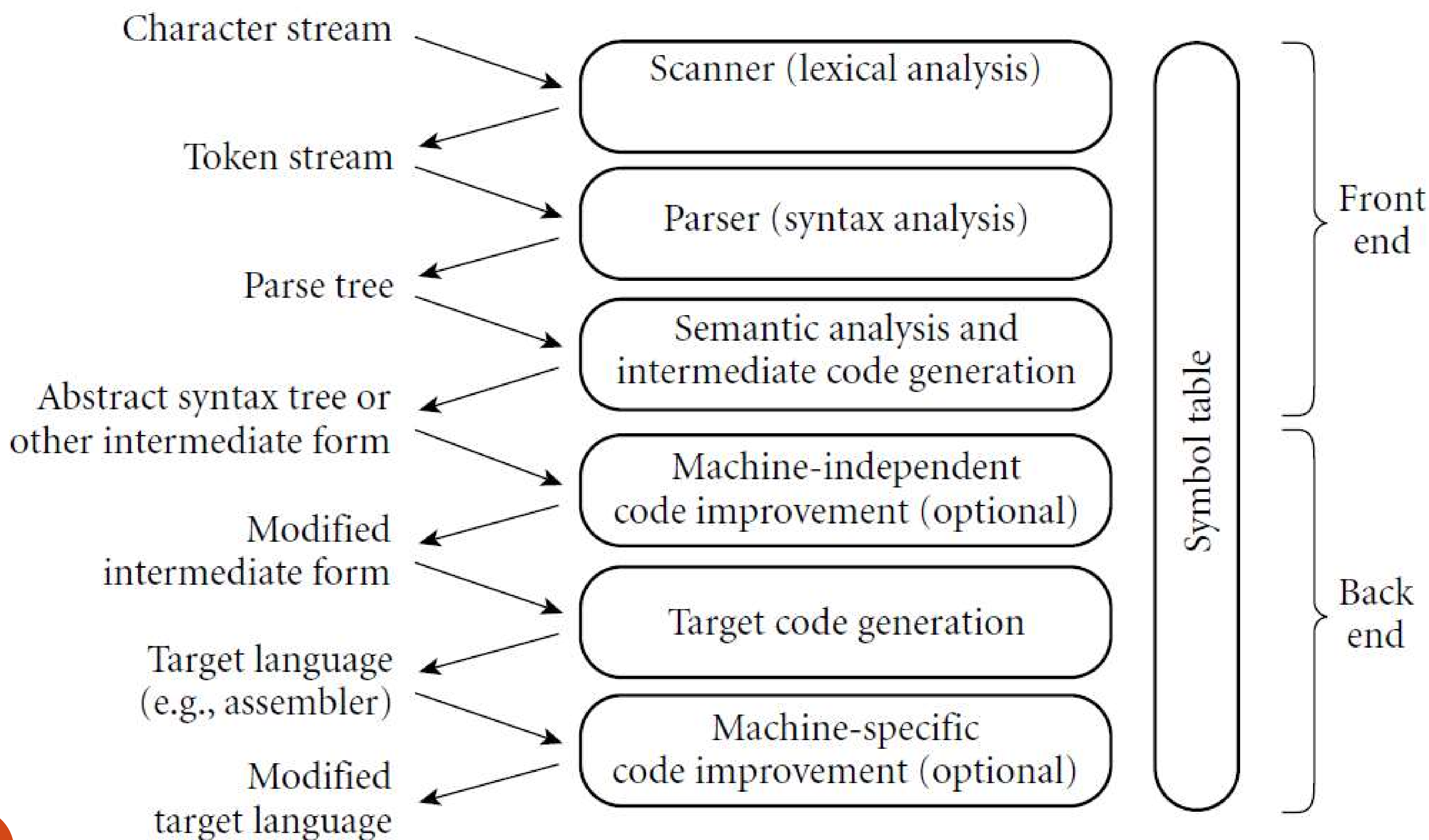
**GNU Assembler**

| | |
|---|---|
| **Developer(s)** | GNU Project |
| **Stable release** | 2.29.1 / September 25, 2017; 10 months ago |
| **Written in** | C |
| **Platform** | Cross-platform |
| **Type** | Assembler |
| **License** | GNU General Public License v3 |
| **Website** | www.gnu.org/software/binutils/ |

The GAS executable is named `as`, the standard name for a Unix assembler. GAS is cross-platform, and both runs on and assembles for a number of different computer architectures. Released under the GNU General Public License v3, GAS is free software.

# 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
  - Organizes tokens in a parse tree

# 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 **<u>resemble machine code for some imaginary idealized machine</u>**; 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 <u>during or after target code generation</u>

# 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*** → *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* → $\epsilon$

and

*block-item-list* → *block-item*

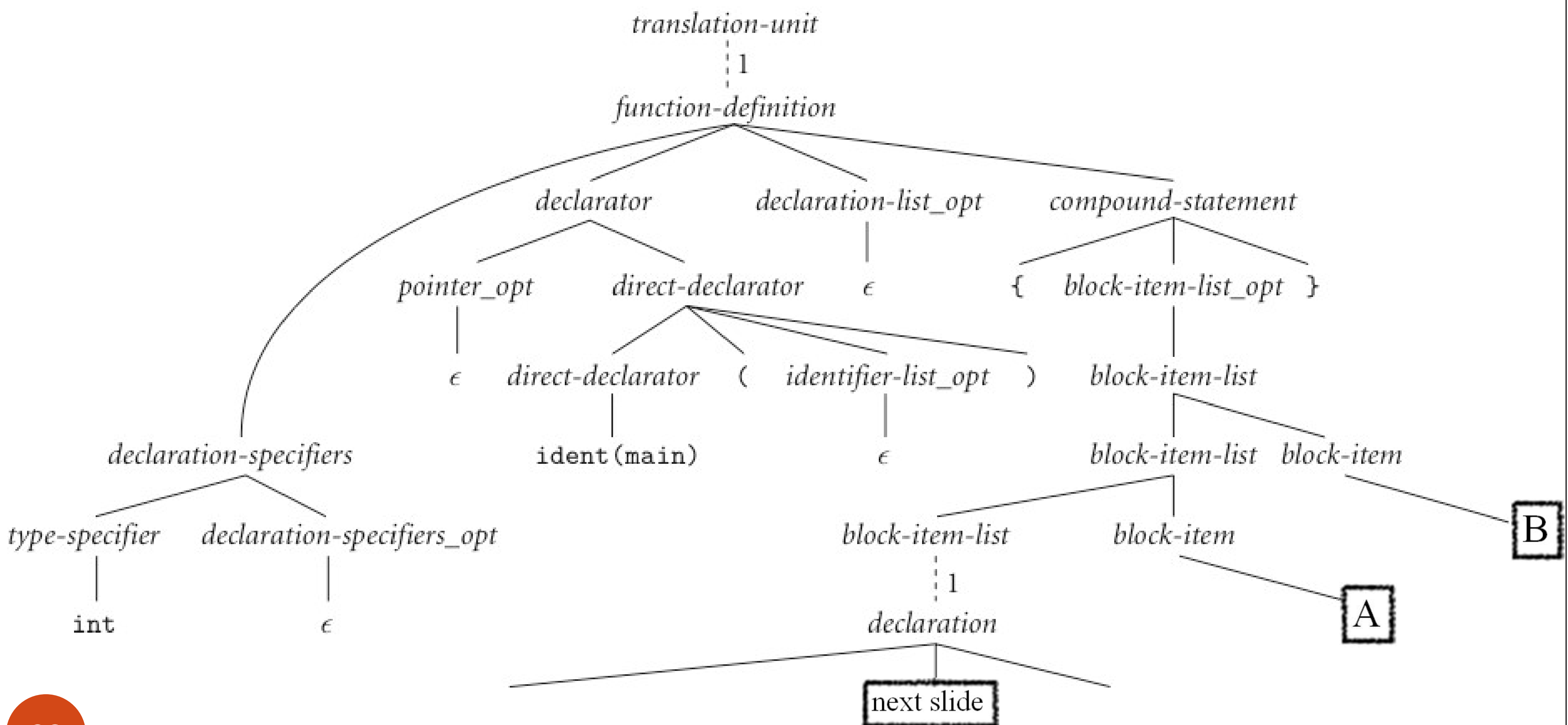
*block-item-list* → *block-item-list block-item*

*block-item* → *declaration*

*block-item* → *statement*

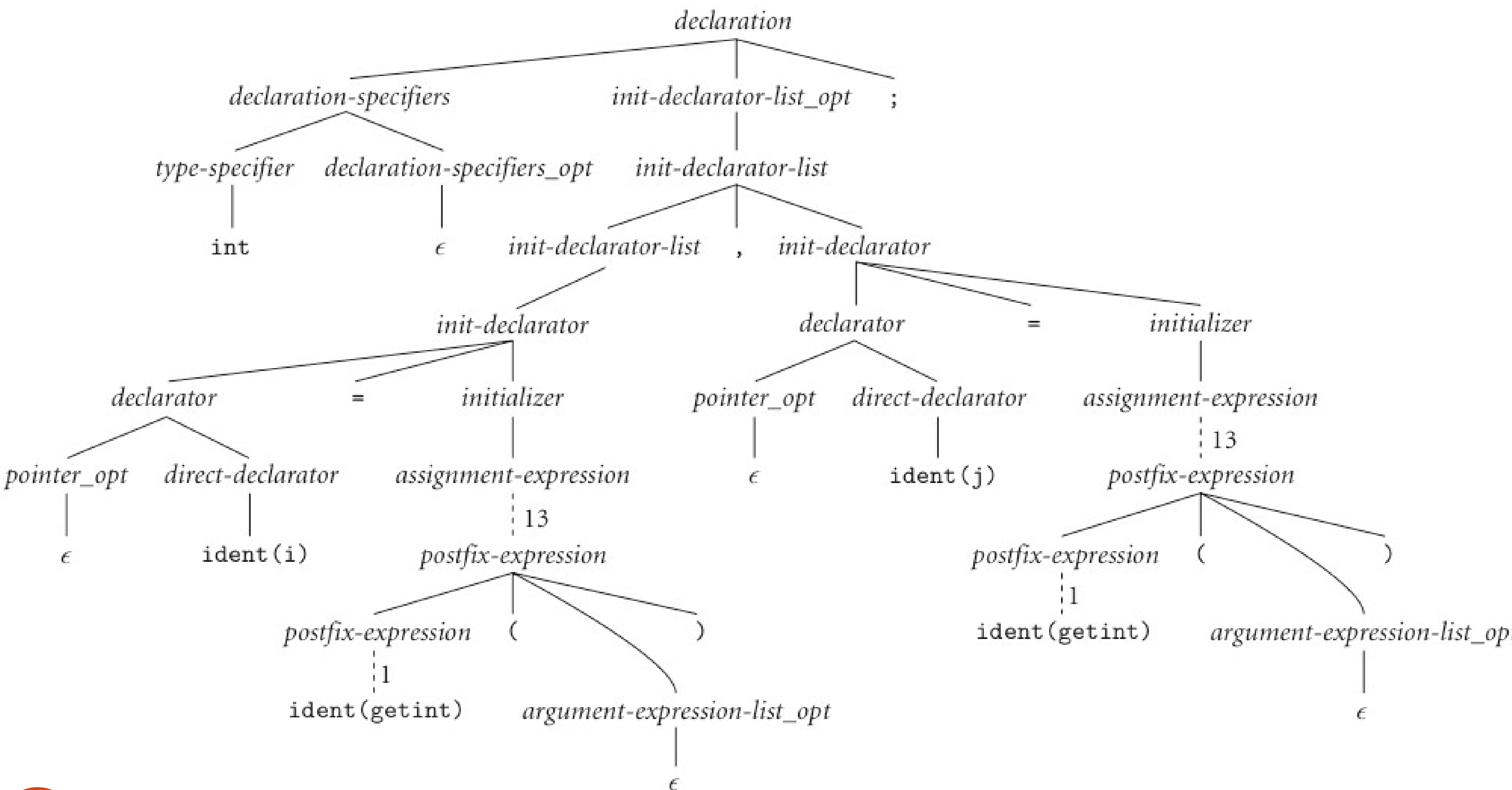
# An Overview of Compilation

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



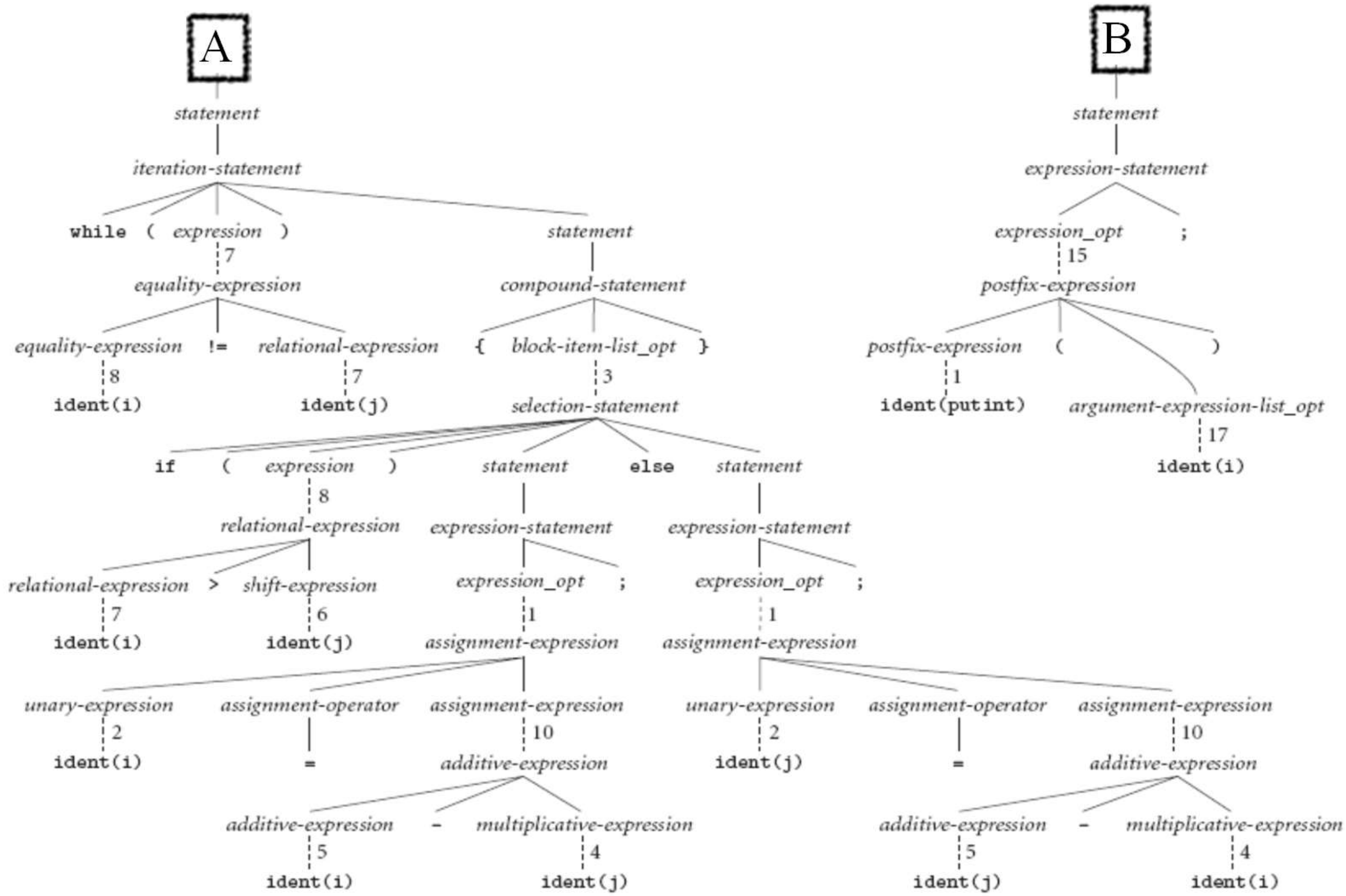
# An Overview of Compilation

- 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

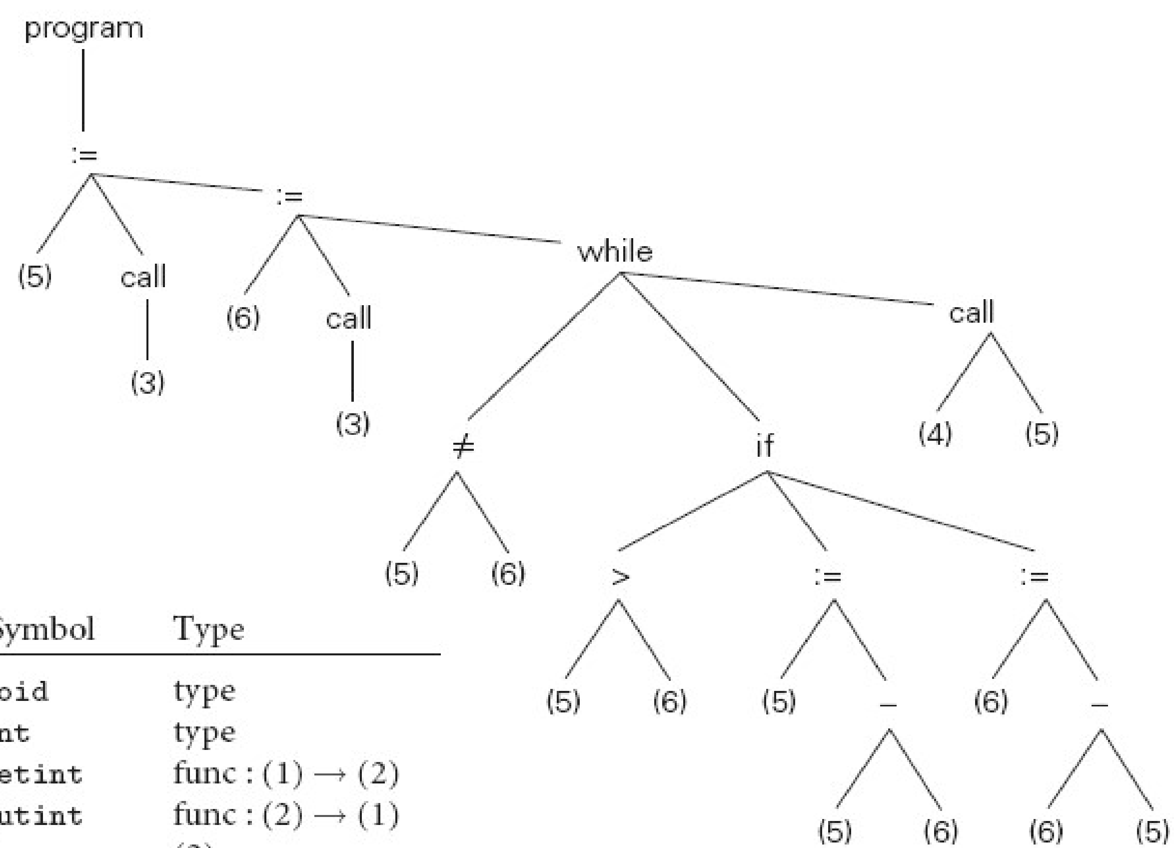
- Semantic analysis implementation
  - *semantic action routines* are invoked by the parser when it realizes that it has reached a particular point within a grammar rule.
- Not all semantic rules can be checked at compile time: only the *static semantics* of the language
  - the *dynamic semantics* of 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

- <u>GCD Syntax Tree (AST)</u>




| 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

- **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, <u>the other immediately after target code generation</u>