Chapter 1 :: Introduction

Programming Language Pragmatics

Introduction

- Why are there so many programming languages?
  - evolution -- we've learned better ways of doing things over time
  - orientation toward special purposes
  - diverse ideas about what is pleasant to use

Introduction

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

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

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, varargs, catch and throw

Why study programming languages?

- Help you make better use of whatever language you use (2)
  - understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
    - Pointers for array traversal in C

Why study programming languages?

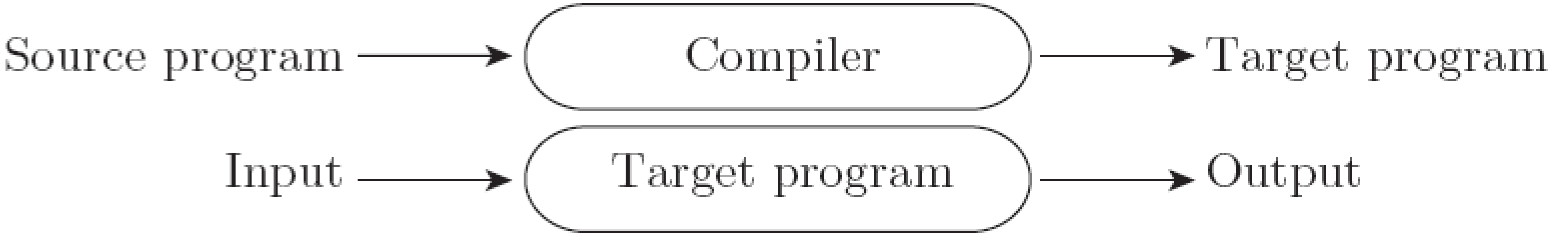
- Help you make better use of whatever language you use (3)
  - figure out how to do things in languages that don't support them explicitly

Imperative languages

- 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)

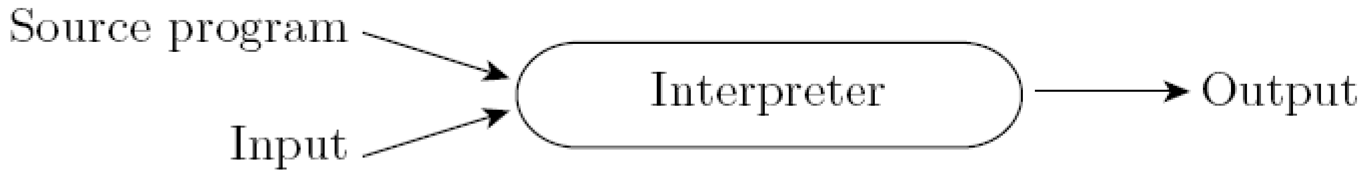
Compilation vs. Interpretation

- 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



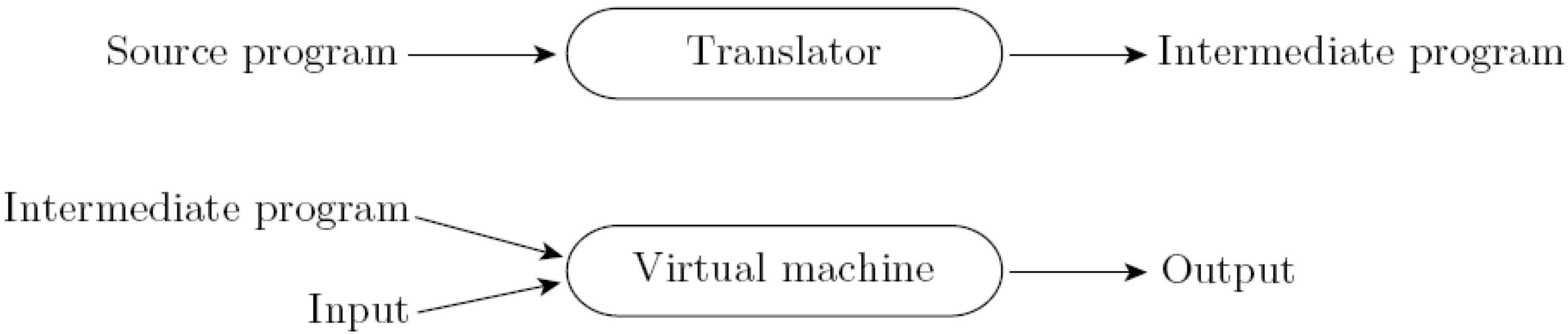
Compilation vs. Interpretation

- 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 language implementations include a mixture of both compilation and interpretation



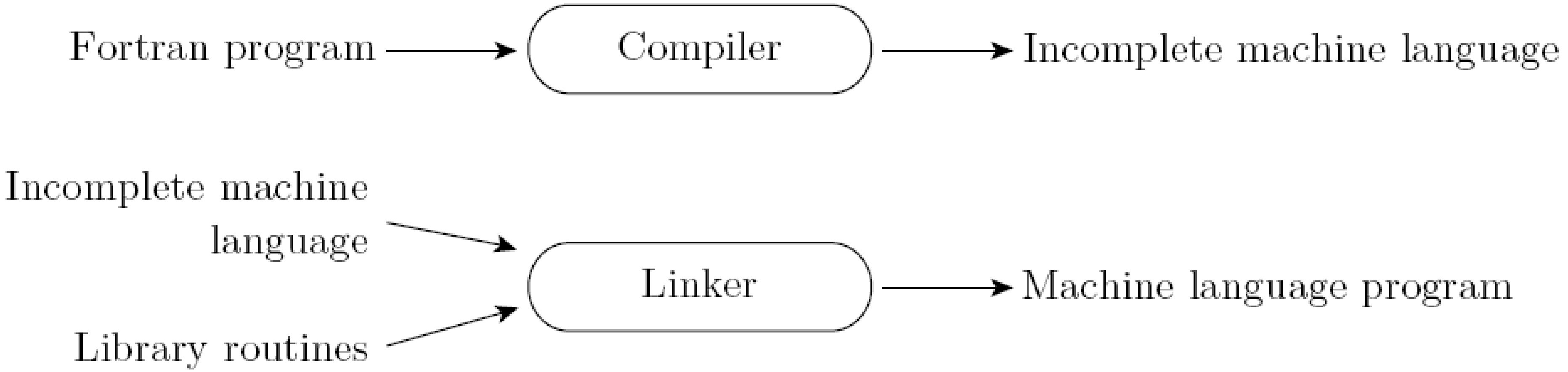
ELSEVIER

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)

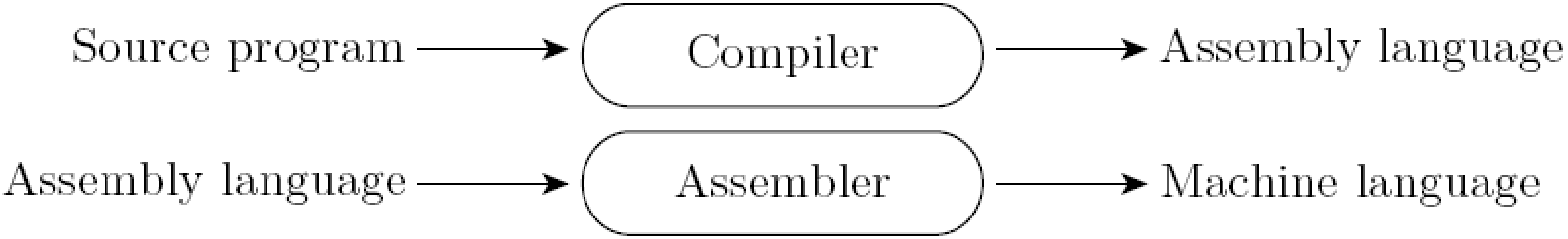
Compilation vs. Interpretation

- Implementation strategies:
  - 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:



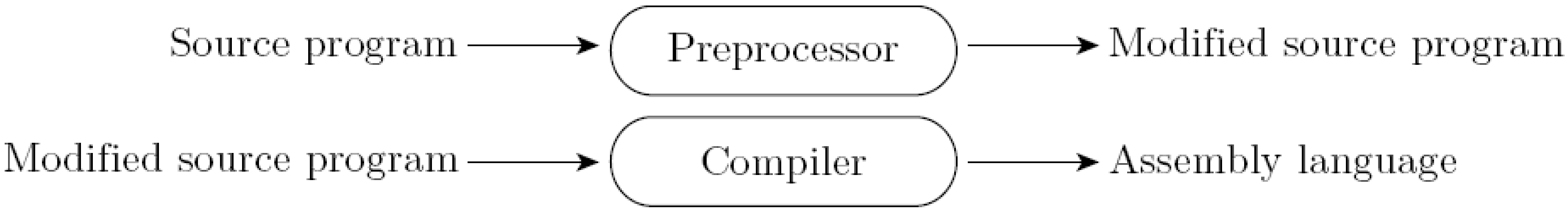
Compilation vs. Interpretation

- 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)



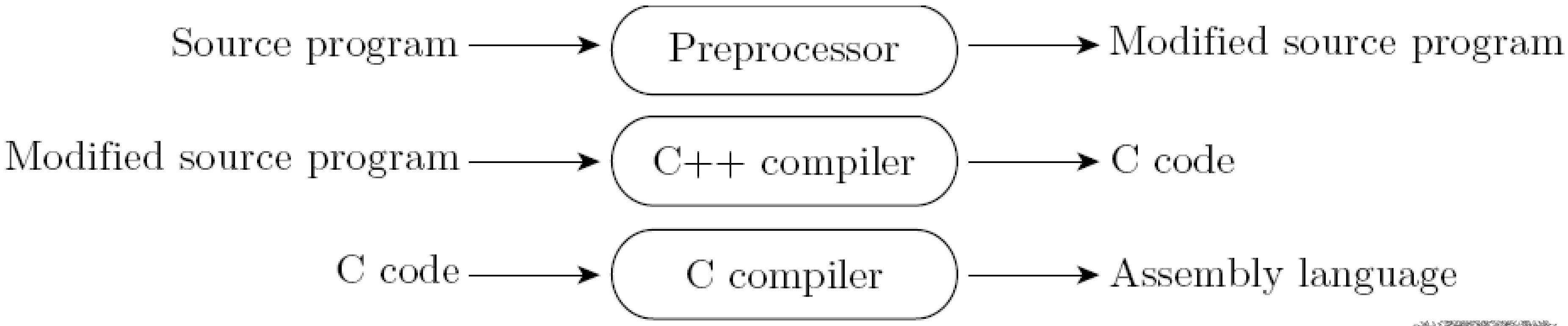
Compilation vs. Interpretation

- Implementation strategies:
  - The C Preprocessor (conditional compilation)
    - Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source



Compilation vs. Interpretation

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



Compilation vs. Interpretation

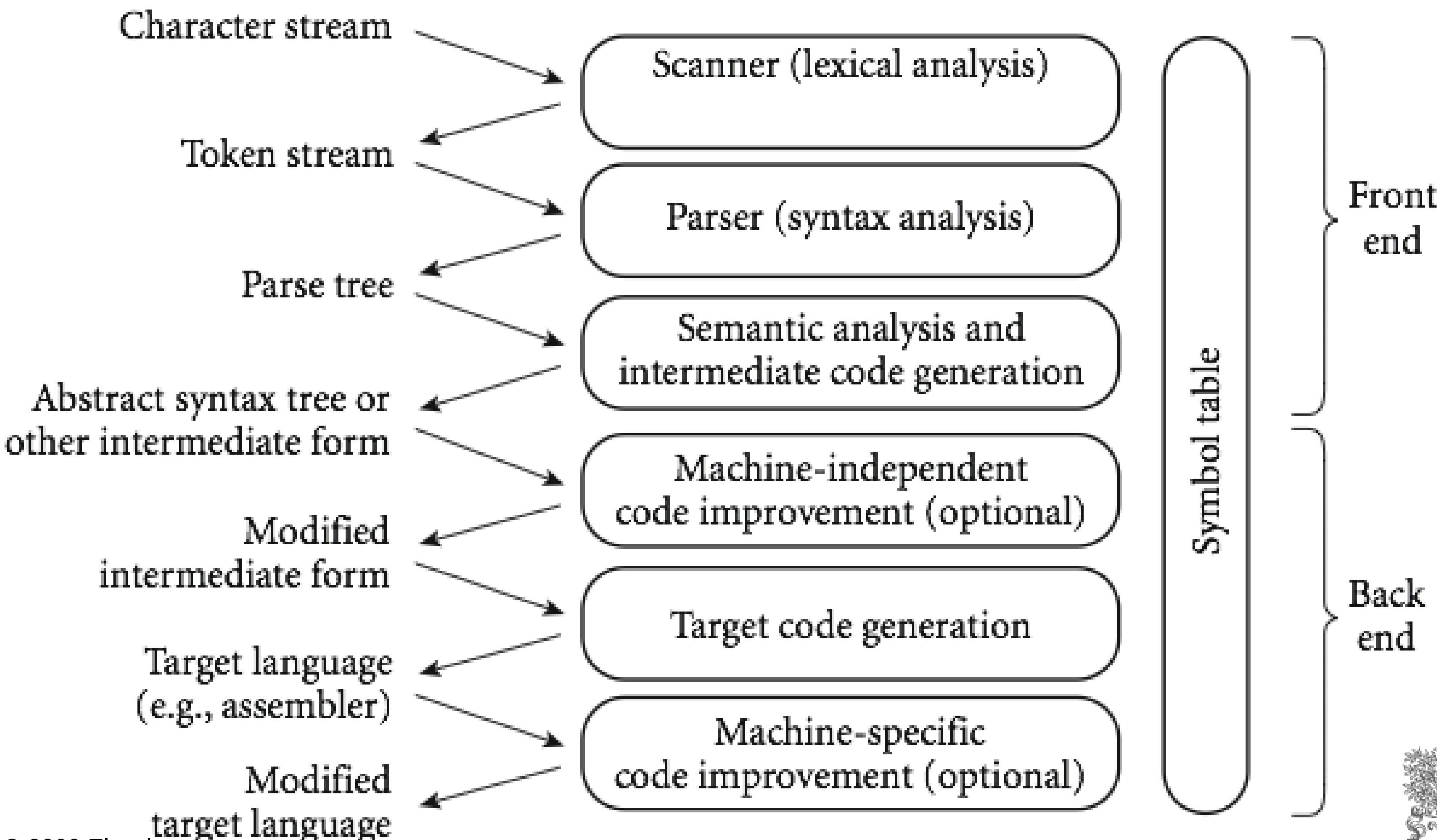
- Implementation strategies:
  - Compilation of Interpreted Languages
    - The compiler generates code that makes assumptions about decisions that won't be finalized until runtime. If these assumptions are valid, the code runs very fast. If not, a dynamic check will revert to the interpreter.

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.
      - The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs.
      - The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

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

An Overview of Compilation

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

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

- ***Optimization*** takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - The term is a misnomer; we just *improve* code
  - The optimization phase is optional
- ***Code generation phase*** produces assembly language or (sometime) relocatable machine language

An Overview of Compilation

- Certain ***machine-specific optimizations*** (use of special instructions or addressing modes, etc.) may be performed during or after ***target code generation***
- ***Symbol table***: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
  - This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

An 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);
}
```

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

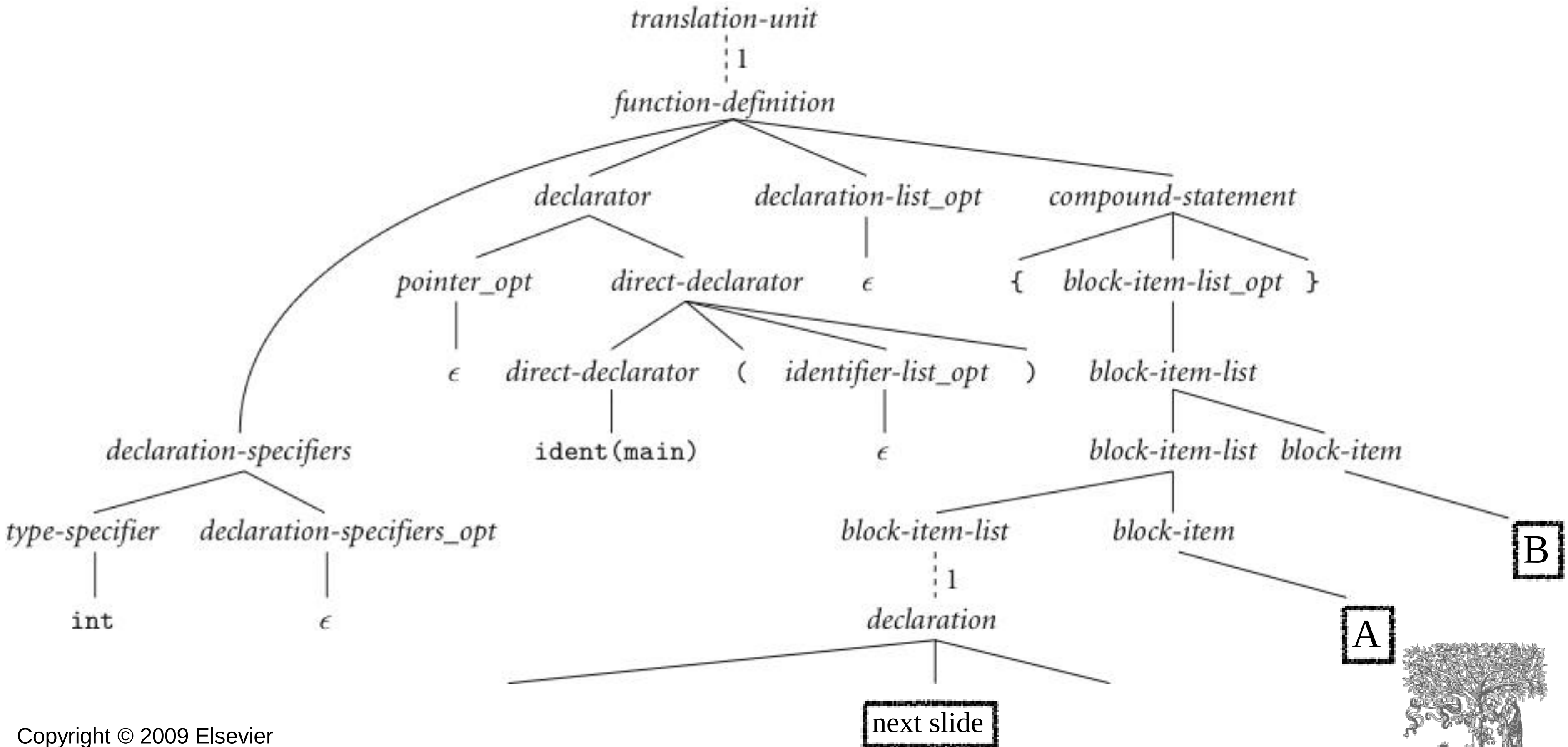
- Context-Free Grammar and Parsing
  - Example (`while` loop in C)

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*

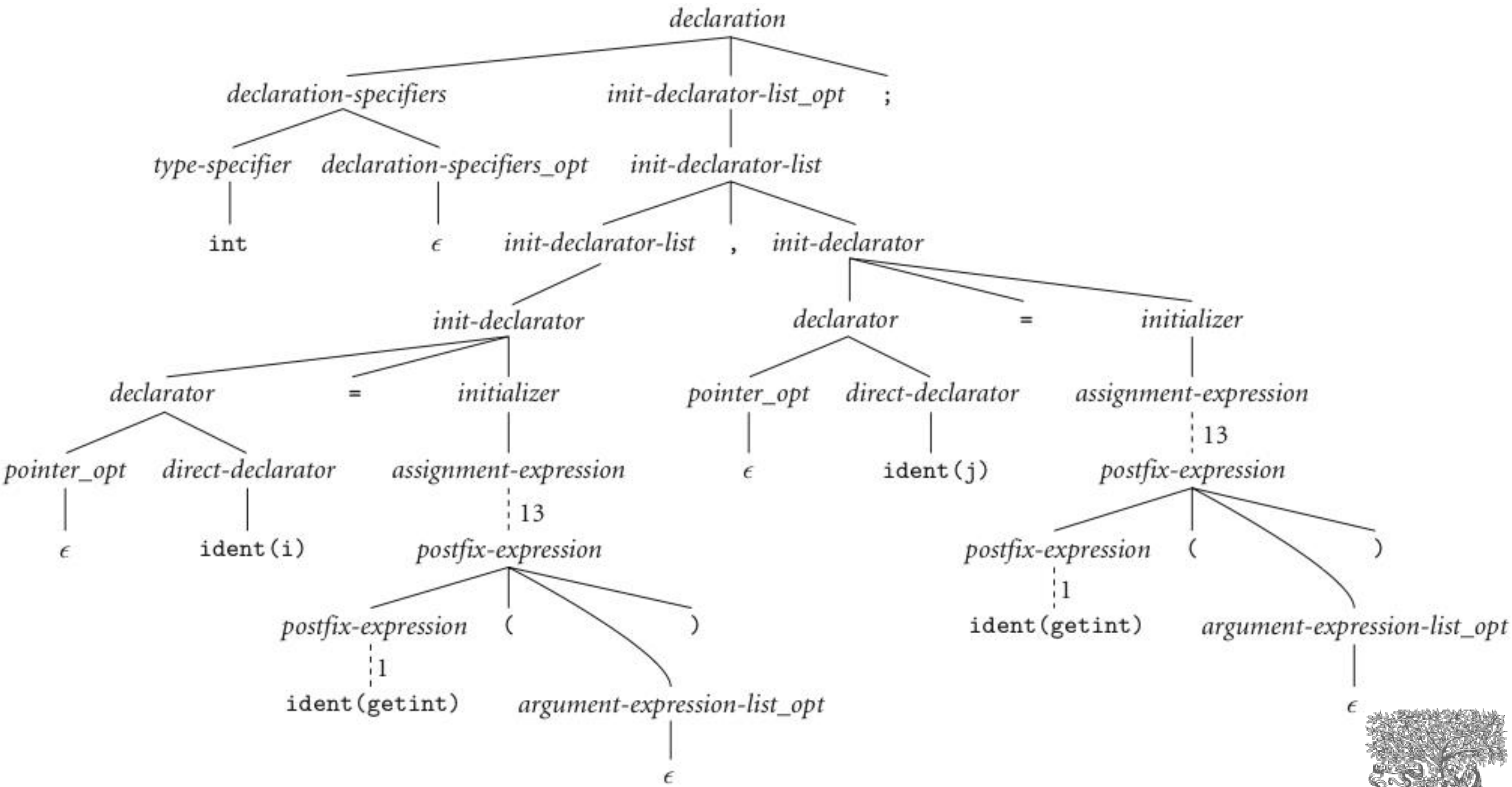
An Overview of Compilation

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



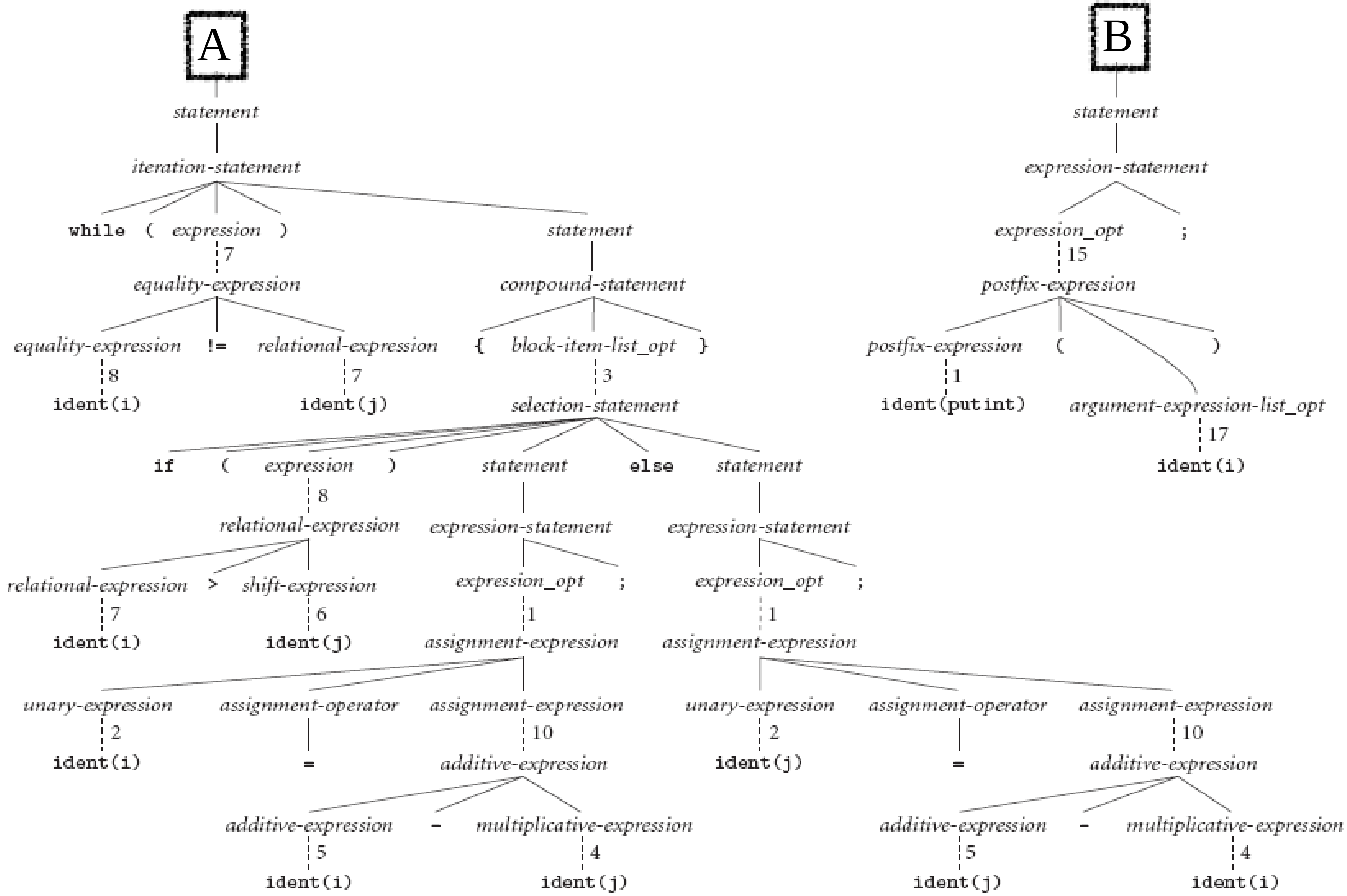
An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



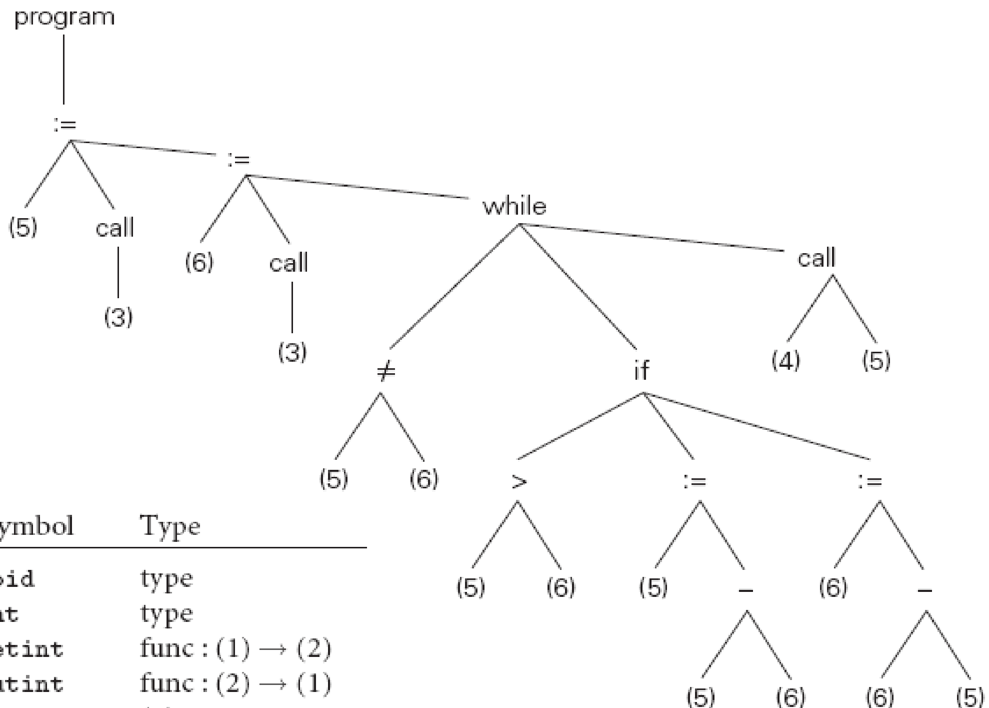
An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



An Overview of Compilation

- 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)