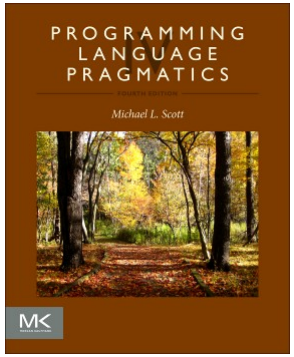
Chapter 1 :: Introduction - Compilation Overview

Programming Language Pragmatics, Fourth Edition

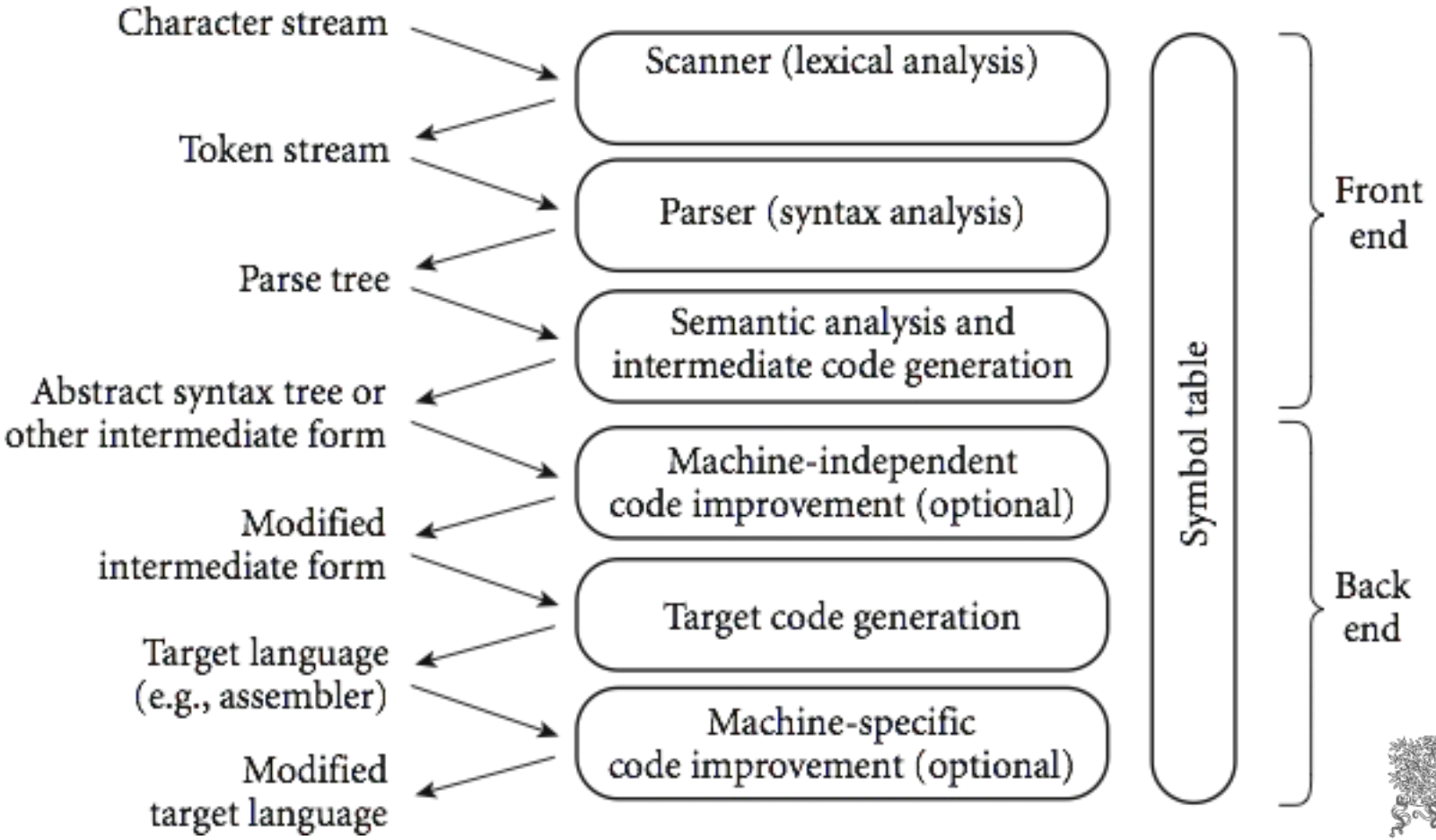
Michael L. Scott



ELSEVIER

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

- ***Intermediate form*** (IF) 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
- ***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

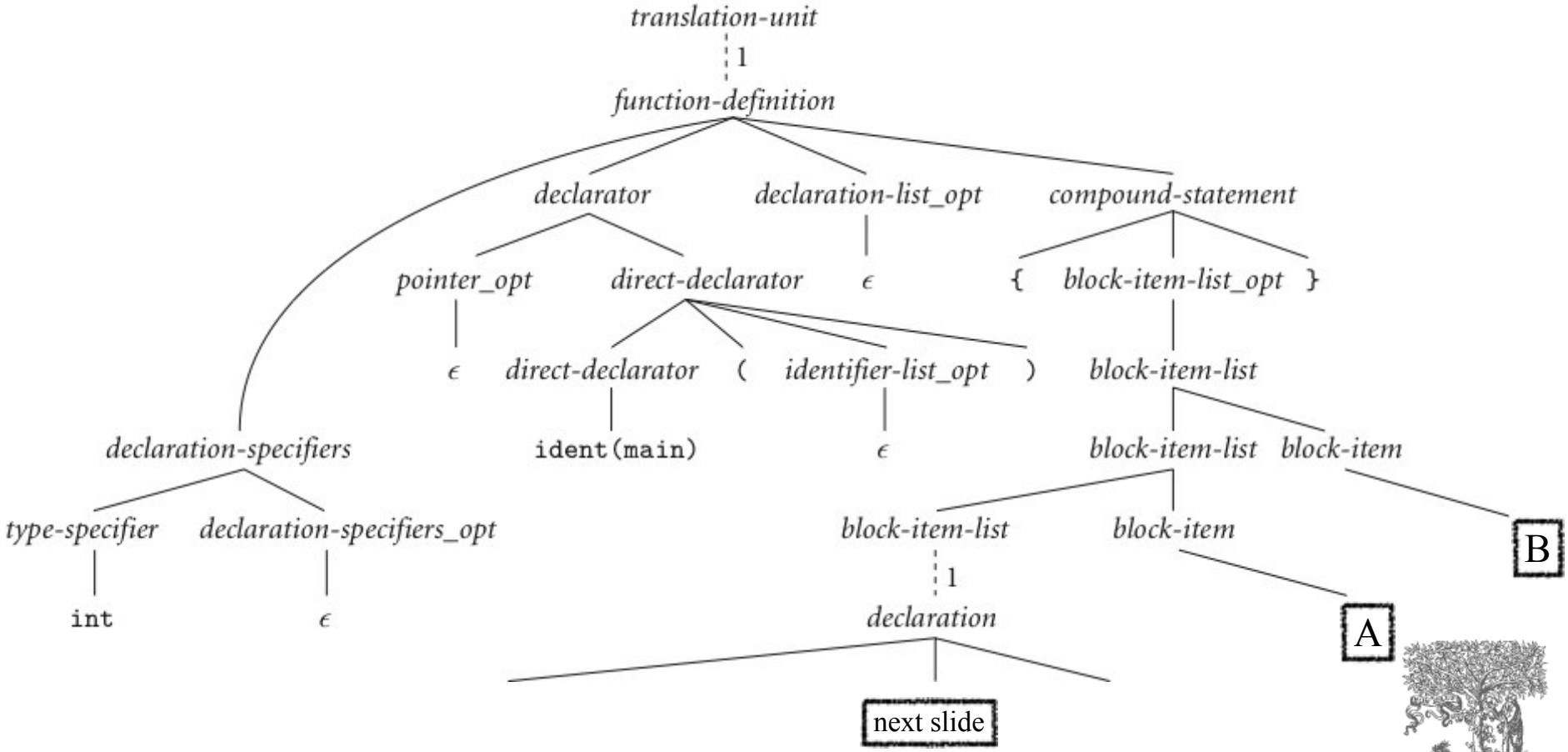
- Context-Free Grammar and Parsing
  - Example (`while` loop in C)

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*

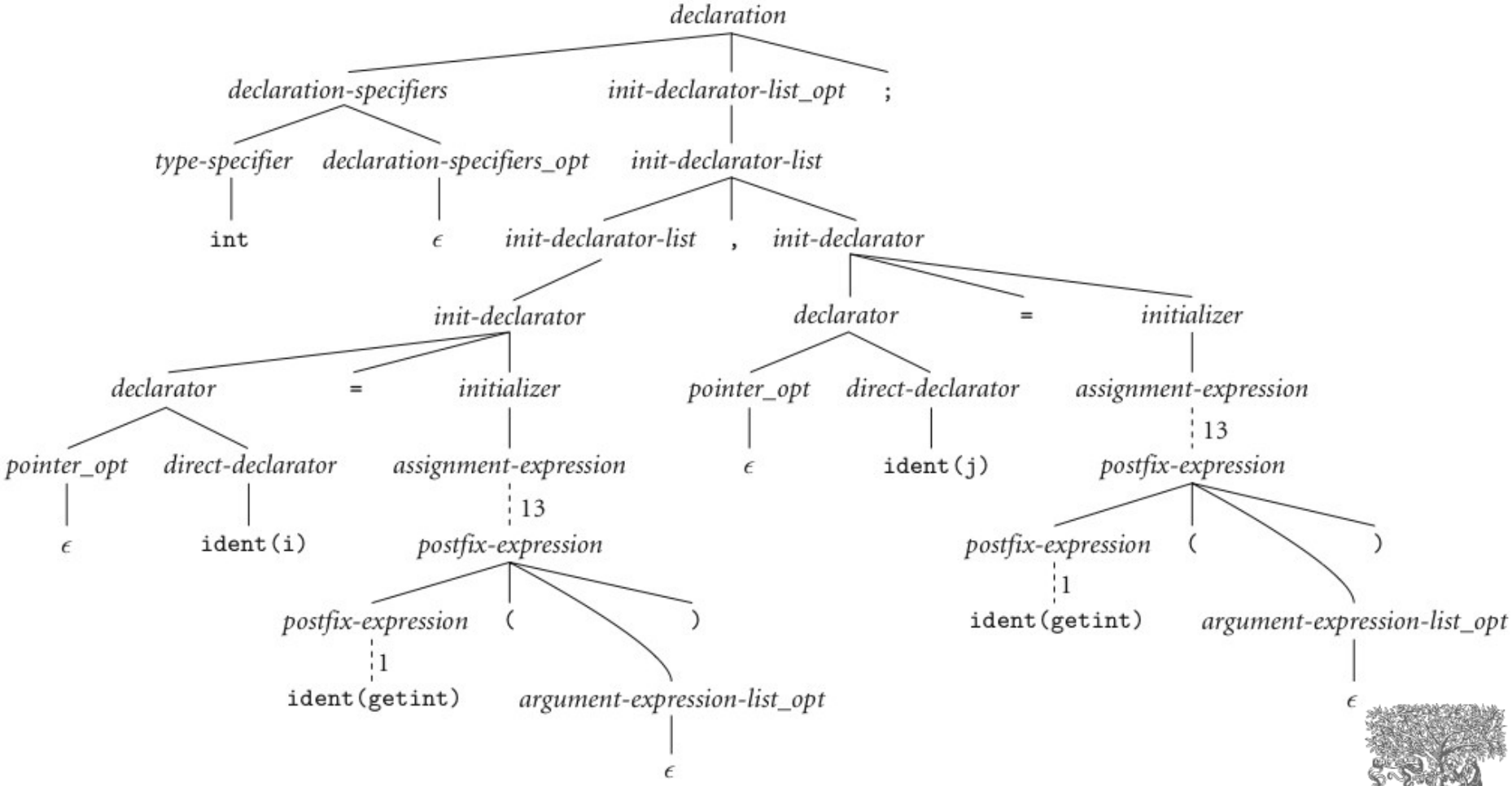
An Overview of Compilation

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



An Overview of Compilation

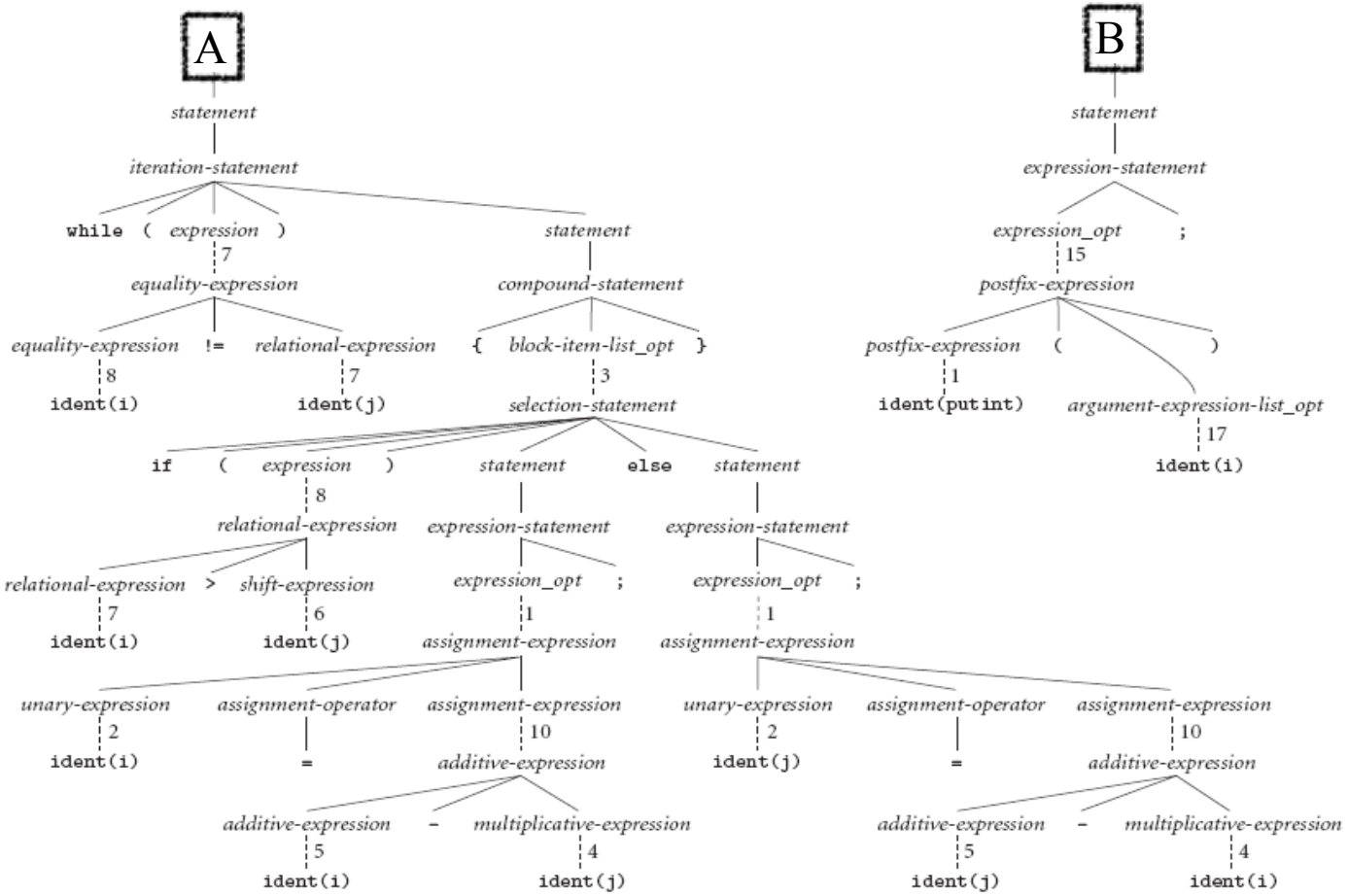
- Context-Free Grammar and Parsing (continued)



ELSEVIER

An Overview of Compilation

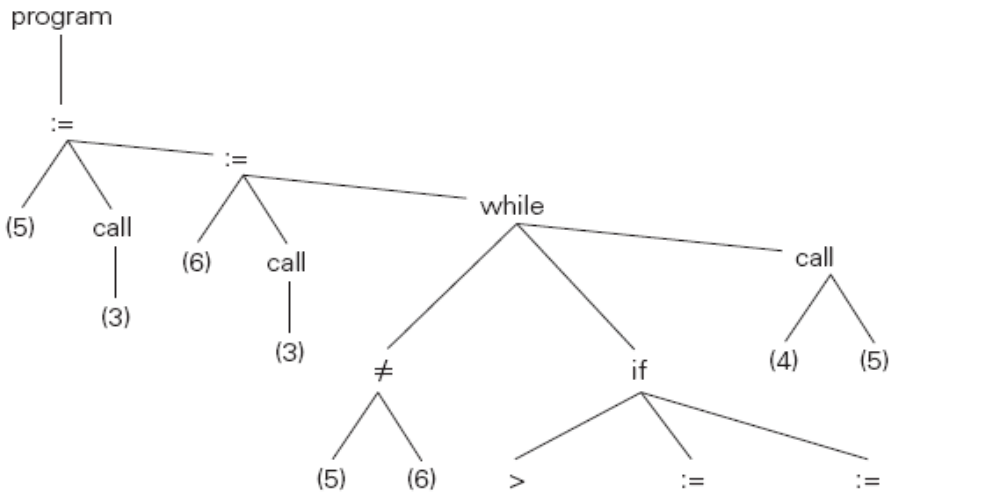
- Context-Free Grammar and Parsing (continued)



ELSEVIER

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)