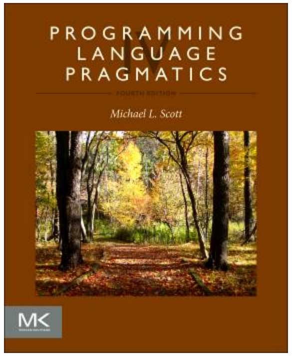
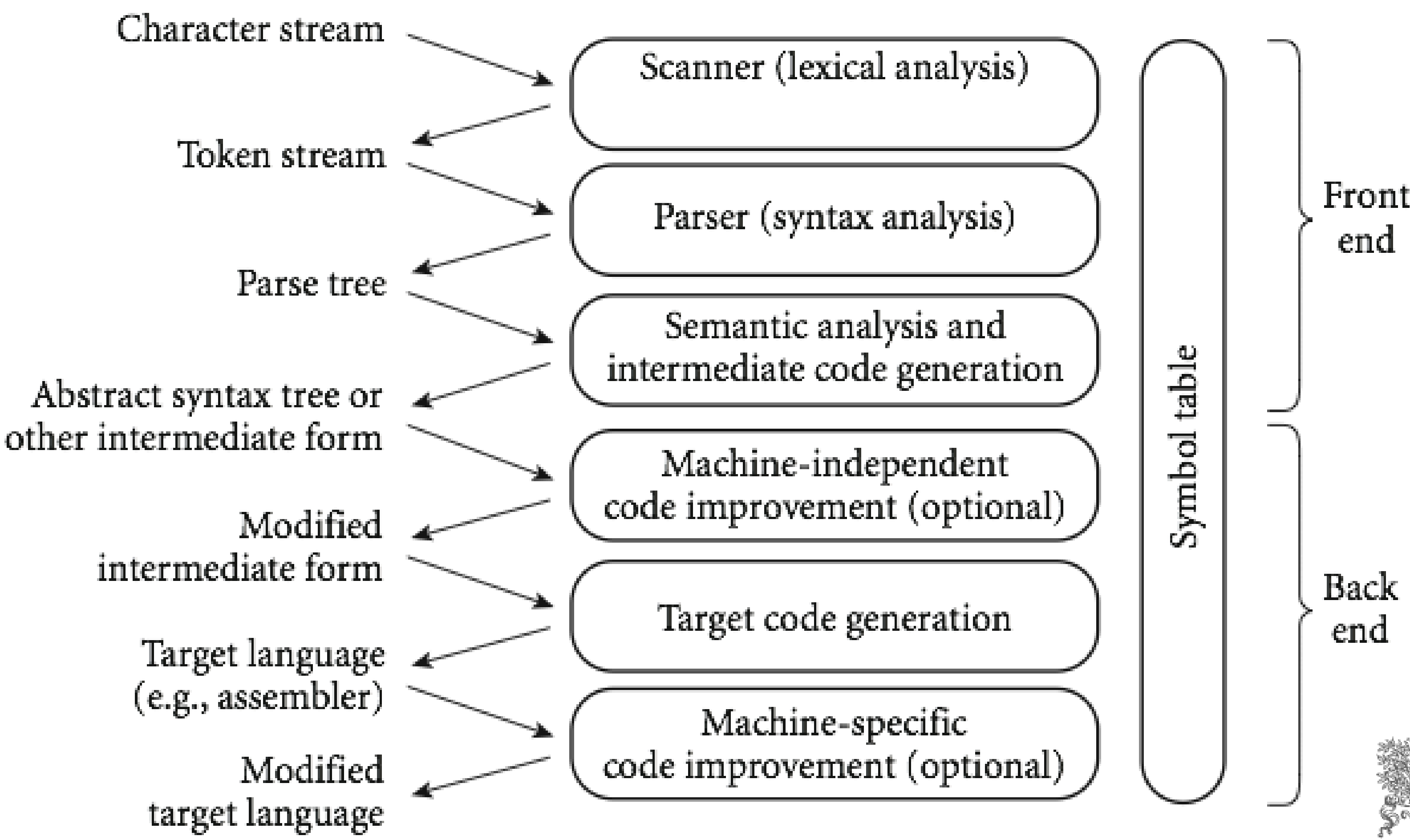
Chapter 1 :: Introduction

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Phases of Compilation



Scanning (Lexical Analysis)

- Scanning divides the program into "tokens", which are the smallest meaningful units.
- In the process of scanning, the compiler checks to see that all of the program's tokens are well formed.
- Scanning is recognition of a *regular language*, e.g., via DFA.
- A scanner is a DFA that recognizes the tokens of a programming language.

ELSEVIER

Parsing (Syntax Analysis)

- In the process of parsing, the compiler checks that the sequence of tokens conforms to the syntax defined by the **context-free grammar**.
- Parsing is recognition of a *context-free language*, e.g., via PDA.
- A parser is a deterministic PDA that recognizes the language's context-free syntax.

Lexical and Syntax Analysis

- Recognize the structure of the program, groups characters into *tokens*
- Serve to recognize the structure of the program, without regard to its meaning

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

GCD program in C

Lexical Analysis

- GCD Program Tokens
  - The scanner reads characters and groups them into tokens, which are the smallest meaningful units of the program.

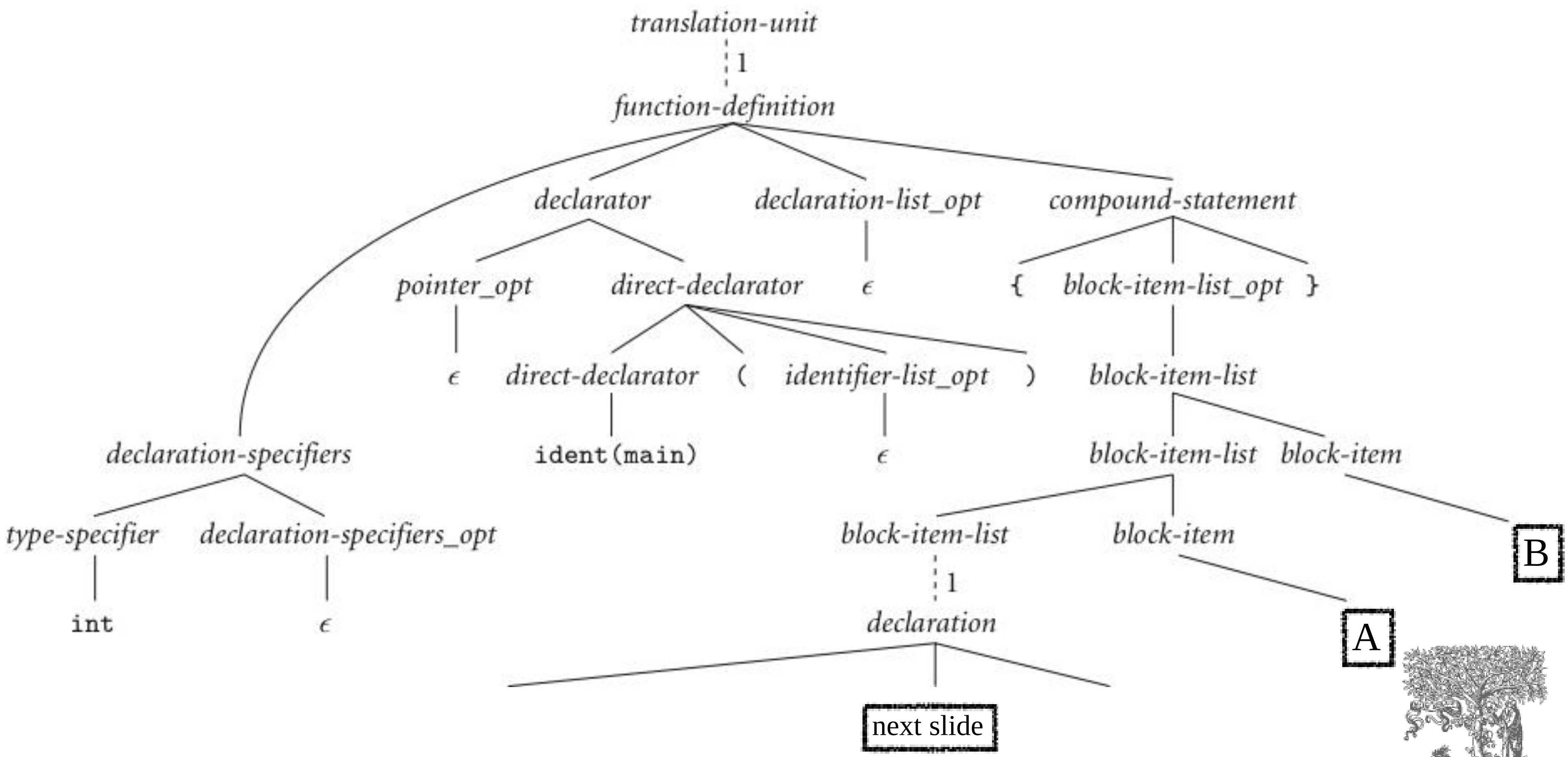
int	main	(	)	{	int	i	=
getint	(	)	,	j	=	getint	(
)	;	while	(	i	!=	j	)
{	if	(	i	>	j	)	i
=	i	-	j	;	else	j	=
j	-	i	;	}	putint	(	i
)	;	}					

Context-Free Grammar and Parsing

- Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents.
- The structure relies on a set of potentially recursive rules known as *context-free grammar* define the ways in which these constituents combine.

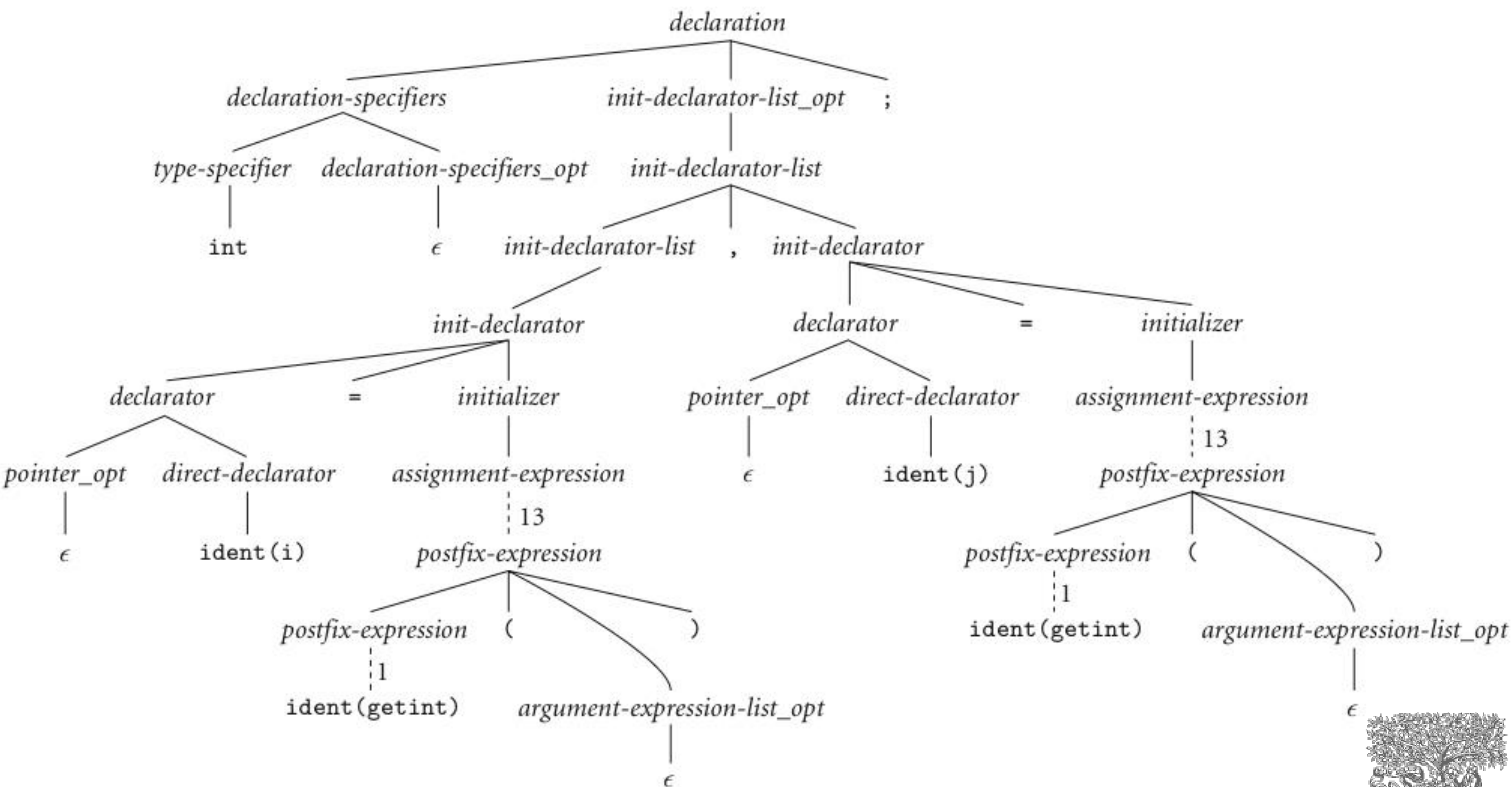
Context-Free Grammar and Parsing

- GCD Program Parse Tree



Context-Free Grammar and Parsing

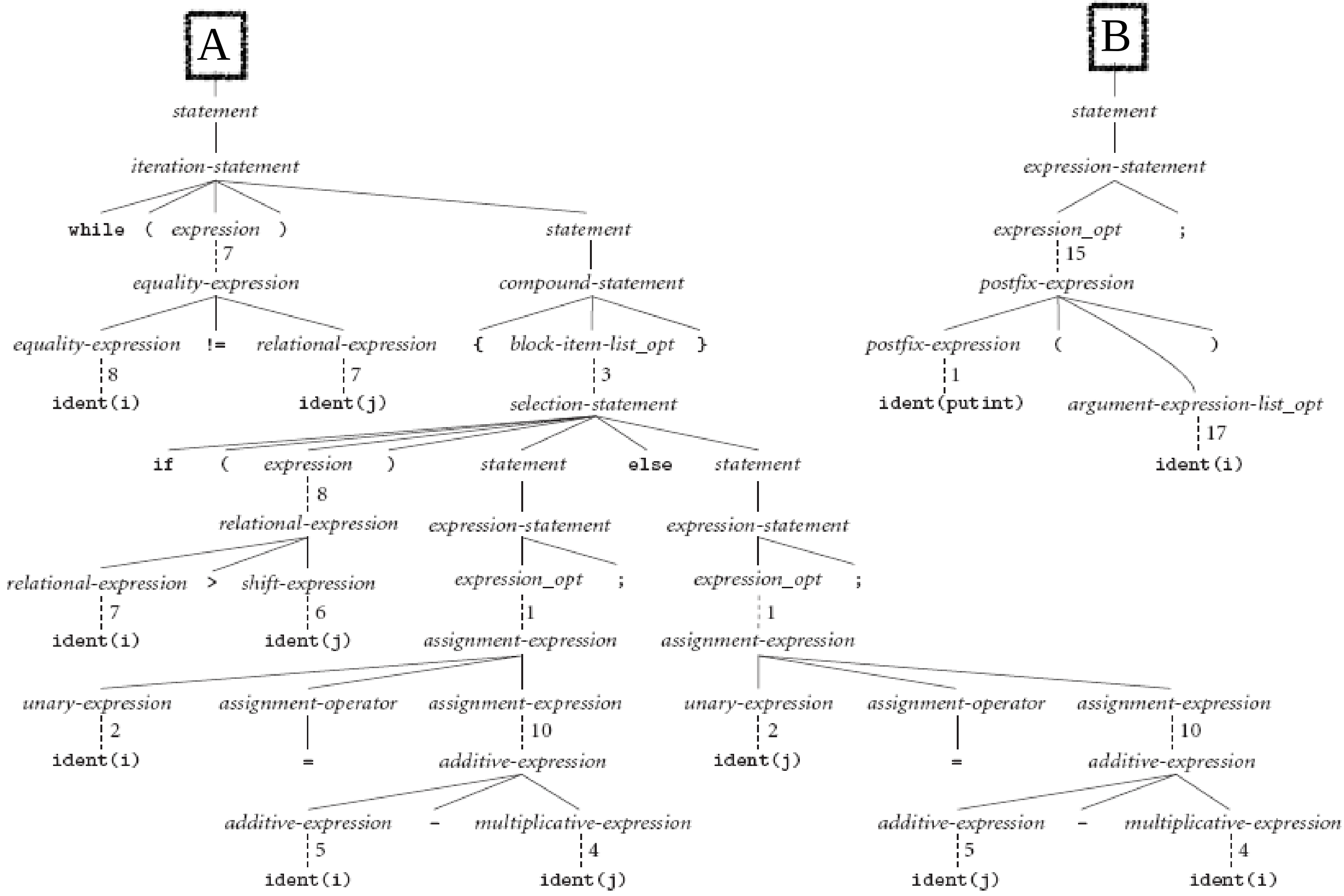
- GCD Program Parse Tree (continued)



ELSEVIER

Context-Free Grammar and Parsing

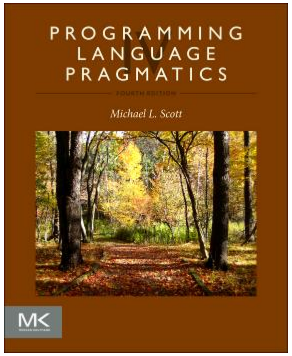
- GCD Program Parse Tree (continued)



Chapter 2 :: Programming Language Syntax

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



ELSEVIER

Introduction

- Computer languages must be precise.
- Both their form (syntax) and meaning (semantics) must be specified without ambiguity.
  - So that both programmers and computers can tell what a program is supposed to do.
- To provide the needed degree of precision, language designers and implementors use formal syntactic and semantic notation.

Introduction

- In this lecture, we focus on syntax:
  1. how we specify the structural rules of a programming language
  2. how a compiler identifies the structure of a given input program
- The first is of interest mainly to programmers, who want to write valid programs.
  – This task relies on *regular expressions* and *context-free grammars*
- The second is of interest mainly to compilers, which need to analyze those programs.
  – This task relies on *scanners* and *parsers*

Tokens

- Basic building blocks of programs—the shortest strings of characters with individual meaning.
- Tokens come in many kinds:
  - Keywords (if, return, struct, etc. in C)
  - Identifiers (my_variable, sizeof, printf, etc. in C)
  - Symbols ({}, (), :, etc.)
  - Constants of various types
- To specify tokens, we use the notation of *regular expressions*.

Regular Expressions

- A regular expression is one of the following:
  - A character
  - The empty string, denoted by ε
  - Two regular expressions concatenated
    - If E_1 and E_2 are regular ex., then E_1E_2 is a regular ex.
  - Two regular expressions separated by a vertical bar (|), meaning any string generated by the first one *or* any string generated by the second one (i.e., or)
  - A regular expression followed by the Kleene star *, meaning the concatenation of zero or more strings generated by the expression in front of the star

Regular Expressions

- Digits are the syntactic building blocks for numbers.
- A natural number is represented by an arbitrary-length (nonempty) string of digits, beginning with a nonzero digit:

$$\begin{aligned}
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{non_zero_digit} &\longrightarrow \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{natural_number} &\longrightarrow \textit{non_zero_digit}\ \textit{digit}\,^{*}
\end{aligned}$$

Context-Free Grammars

- Regular expressions
  - work well for defining tokens
  - are unable to specify *nested* constructs, which are central to programming languages
- The notation for context-free grammars (CFG) is sometimes called Backus-Naur Form (BNF)

Context-Free Grammars

- A CFG consists of
  - A set of *productions*
    - (each of the rules in a CFG)
  - A set of *non-terminals* (or *variables*) *N*
    - (the symbols on the left-hand sides of the productions)
  - A set of *terminals* *T*
    - (the symbols that are to make up the strings derived from the grammar)
  - A *start symbol* *S*
    - (one of the nonterminals, usually the one on the left-hand side of the first production)

Context-Free Grammars

- Expression grammar with precedence and associativity:

$$
\begin{array}{lllll}
1. & expr & \longrightarrow & term \mid expr\ add\_op\ term \\
2. & term & \longrightarrow & factor \mid term\ mult\_op\ factor \\
3. & factor & \longrightarrow & \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)} \\
4. & add\_op & \longrightarrow & \texttt{+} \mid \texttt{-} \\
5. & mult\_op & \longrightarrow & \texttt{*} \mid \texttt{/}
\end{array}
$$

- A context-free grammar shows us how to generate a syntactically valid string of terminals.

ELSEVIER

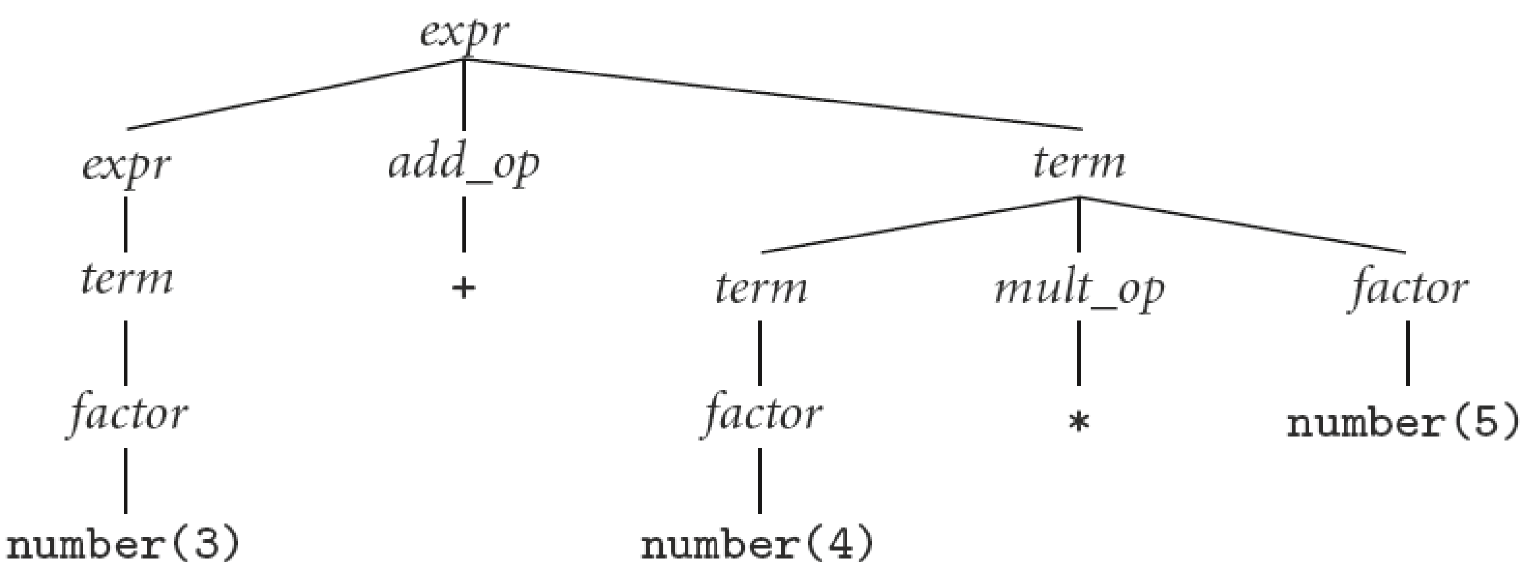
Context-Free Grammars

- Precedence tells us that multiplication and division in most languages group more tightly than addition and subtraction
  - 3 + 4 * 5 means 3 + (4 * 5) rather than (3 + 4) * 5

- Associativity tells us that the operators in most languages group left to right:
  - 10 - 4 - 3 means (10 - 4) - 3 rather than 10 - (4 - 3)

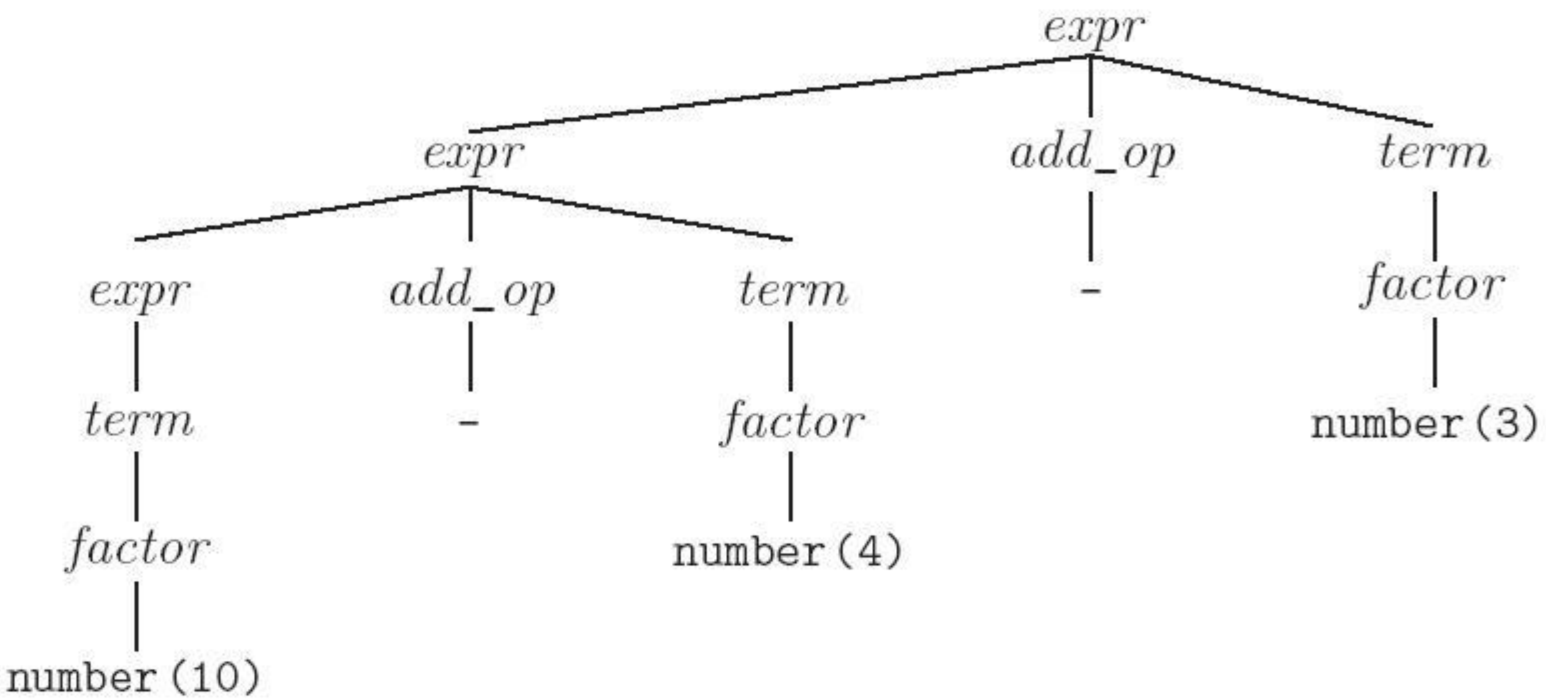
Context-Free Grammars

- Parse tree for expression grammar (with precedence) for
  `3 + 4 * 5`



Context-Free Grammars

- Parse tree for expression grammar (with left associativity) for **10 − 4 − 3**



Parsing

- Terminology:
  - context-free grammar (CFG)
  - symbols
    - terminals (tokens)
    - non-terminals
  - production
  - derivations (left-most and right-most - canonical)
    - A series of replacement operations that shows how to derive a string of terminals from the start symbol is called a derivation.
  - parse trees
    - A parse tree is a hierarchical representation of a derivation.
    - We can represent a derivation graphically as a parse tree.

Parsing

- By analogy to RE and DFAs, a context-free grammar (CFG) is a *generator* for a context-free language (CFL)
  - a parser is a language *recognizer*

Parsing

- There are large classes of grammars for which we can build parsers that run in linear time
  - The two most important classes are called **LL** and **LR**
- LL stands for "Left-to-right, Leftmost derivation"
- LR stands for "Left-to-right, Rightmost derivation"

Parsing

- LL parsers are also called "top-down", or "predictive" parsers
- LR parsers are also called "bottom-up", or "shift-reduce" parsers

Class	Direction of scanning	Derivation discovered	Parse tree construction	Algorithm used
LL	left-to-right	left-most	top-down	predictive
LR	left-to-right	right-most	bottom-up	shift-reduce

- In both the input is read left-to-right, and the parser attempts to discover (construct) a derivation of that input.
  - For LL parsers, the derivation will be left-most
  - For LR parsers, the derivation will be right-most

Parsing

- You commonly see LL or LR (or whatever) written with a number in parentheses after it
  - This number indicates how many tokens of look-ahead are required in order to parse.
  - Almost all real compilers use one token of look-ahead.

LL Parsing

- Here is an LL(1) grammar (Fig 2.16):

```
1.  program          → stmt_list $$
2.  stmt_list        → stmt stmt_list
3.                   | ε
4.  stmt     →      id := expr
5.                   | read id
6.                   | write expr
7.  expr     →      term term_tail
8.  term_tail → add op term term_tail
9.                   | ε
```

LL Parsing

- LL(1) grammar (continued)

```
10. term    →     factor fact_tailt
11. fact_tail → mult_op fact fact_tail
12.                 | ε
13.  factor  →      ( expr )
14.                 | id
15.                 | number
16.  add_op →       +
17.                 | -
18.  mult_op →      *
19.                 | /
```

LL Parsing

- Example (average program)

```
read A
read B
sum := A + B
write sum
write sum / 2
```

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …		

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-

LL Parsing

- Here is an LL(1) grammar (Fig 2.16):

```
1.  program          → stmt_list $$
2.  stmt_list        → stmt stmt_list
3.                   | ε
4.  stmt     →       id := expr
5.                   | read id
6.                   | write expr
7.  expr     →       term term_tail
8.  term_tail → add op term term_tail
9.                   | ε
```

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1

LL Parsing

- Here is an LL(1) grammar (Fig 2.16):

```
1.  program          → stmt_list $$
2.  stmt_list        → stmt stmt_list
3.                   | ε
4.  stmt     →         id := expr
5.                   | read id
6.                   | write expr
7.  expr     →         term term_tail
8.  term_tail → add op term term_tail
9.                   | ε
```

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2

LL Parsing

- Here is an LL(1) grammar (Fig 2.16):

```
1.  program          → stmt_list $$
2.  stmt_list        → stmt stmt_list
3.                   | ε
4.  stmt     →       id := expr
5.                   | read id
6.                   | write expr
7.  expr     →       term term_tail
8.  term_tail → add op term term_tail
9.                   | ε
```

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5
read B sum:=A+B …	stmt_list $$	

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5
read B sum:=A+B …	stmt_list $$	
read B sum:=A+B …	stmt stmt_list $$	2

LL Parsing

- Here is an LL(1) grammar (Fig 2.16):

```
1.  program            → stmt_list $$
2.  stmt_list          → stmt stmt_list
3.                     | ε
4.  stmt      →        id := expr
5.                     | read id
6.                     | write expr
7.  expr      →        term term_tail
8.  term_tail → add op term term_tail
9.                     | ε
```

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5
read B sum:=A+B …	stmt_list $$	
read B sum:=A+B …	stmt stmt_list $$	2
read B sum:=A+B …	read id stmt_list $$	5

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5
read B sum:=A+B …	stmt_list $$	
read B sum:=A+B …	stmt stmt_list $$	2
read B sum:=A+B …	read id stmt_list $$	5
sum:=A+B …	stmt_list $$	

LL Parsing

- Here is an LL(1) grammar (Fig 2.16):

```
1.  program           → stmt_list $$
2.  stmt_list         → stmt stmt_list
3.                    | ε
4.  stmt     →        id := expr
5.                    | read id
6.                    | write expr
7.  expr     →        term term_tail
8.  term_tail → add op term term_tail
9.                    | ε
```

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5
read B sum:=A+B …	stmt_list $$	
read B sum:=A+B …	stmt stmt_list $$	2
read B sum:=A+B …	read id stmt_list $$	5
sum:=A+B …	stmt_list $$	
sum:=A+B …	stmt stmt_list $$	2

LL Parsing

- Here is an LL(1) grammar (Fig 2.16):

```
1.  program         → stmt_list $$
2.  stmt_list       → stmt stmt_list
3.                  | ε
4.  stmt     →      id := expr
5.                  | read id
6.                  | write expr
7.  expr     →      term term_tail
8.  term_tail → add op term term_tail
9.                  | ε
```

LL Parsing

Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5
read B sum:=A+B …	stmt_list $$	
read B sum:=A+B …	stmt stmt_list $$	2
read B sum:=A+B …	read id stmt_list $$	5
sum:=A+B …	stmt_list $$	
sum:=A+B …	stmt stmt_list $$	2
sum:=A+B …	id:=expr stmt_list $$	4

LL Parsing

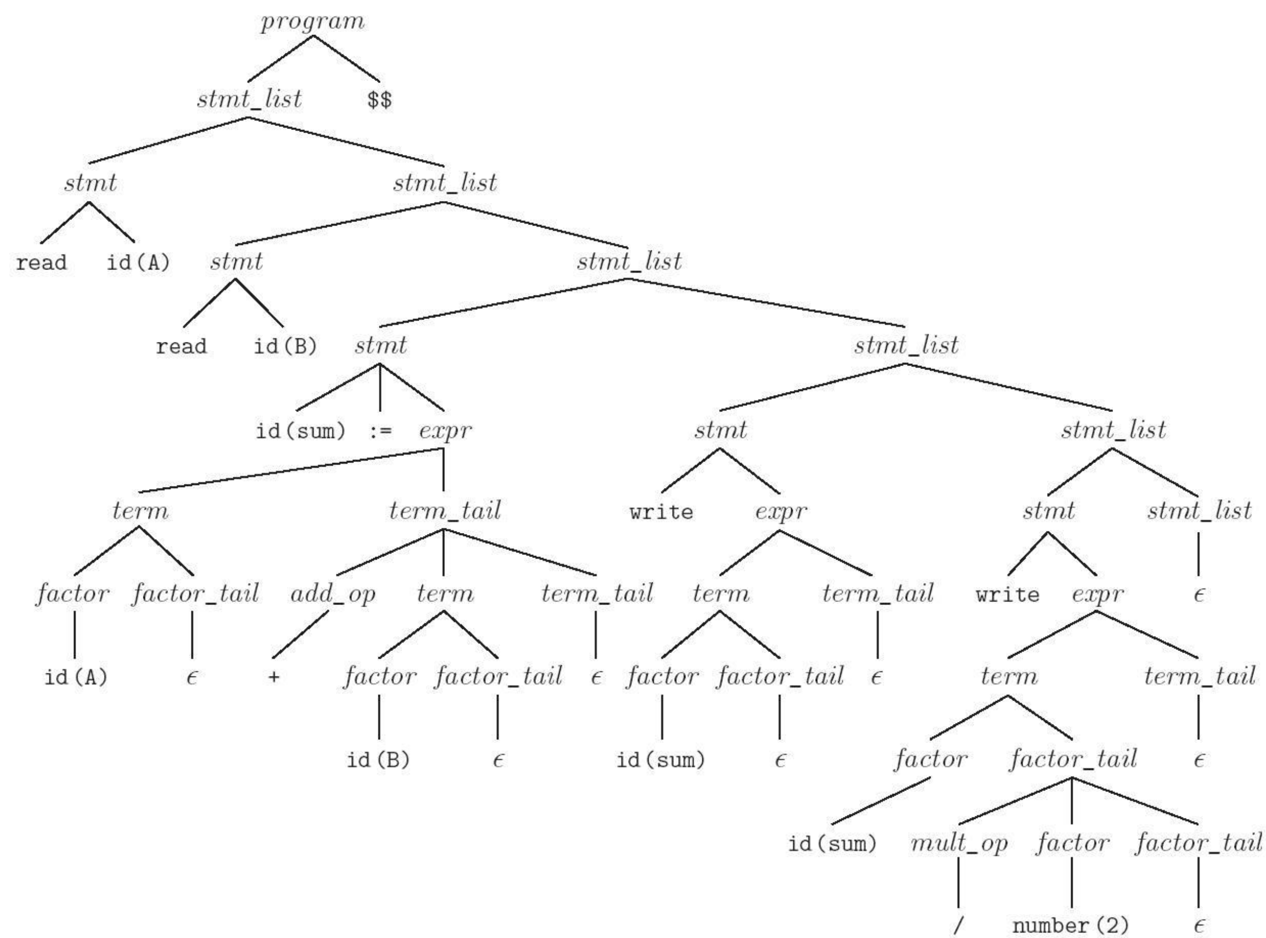
Input	Stack	Rule
read A read B sum:=A+B …	program	-
read A read B sum:=A+B …	stmt_list $$	1
read A read B sum:=A+B …	stmt stmt_list $$	2
read A read B sum:=A+B …	read id stmt_list $$	5
read B sum:=A+B …	stmt_list $$	
read B sum:=A+B …	stmt stmt_list $$	2
read B sum:=A+B …	read id stmt_list $$	5
sum:=A+B …	stmt_list $$	
sum:=A+B …	stmt stmt_list $$	2
sum:=A+B …	id:=expr stmt_list $$	4
…	…	…

Parse stack	Input stream	Comment
program	`read A read B ...`	initial stack contents
stmt_list `$$`	`read A read B ...`	predict *program* $\longrightarrow$ *stmt_list* `$$`
stmt stmt_list `$$`	`read A read B ...`	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read A read B ...`	predict *stmt* $\longrightarrow$ `read id`
`id` *stmt_list* `$$`	`A read B ...`	match `read`
stmt_list `$$`	`read B sum := ...`	match `id`
stmt stmt_list `$$`	`read B sum := ...`	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read B sum := ...`	predict *stmt* $\longrightarrow$ `read id`
`id` *stmt_list* `$$`	`B sum := ...`	match `read`
stmt_list `$$`	`sum := A + B ...`	match `id`
stmt stmt_list `$$`	`sum := A + B ...`	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
`id :=` *expr stmt_list* `$$`	`sum := A + B ...`	predict *stmt* $\longrightarrow$ `id :=` *expr*
`:=` *expr stmt_list* `$$`	`:= A + B ...`	match `id`
expr stmt_list `$$`	`A + B ...`	match `:=`
term term_tail stmt_list `$$`	`A + B ...`	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`A + B ...`	predict *term* $\longrightarrow$ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`A + B ...`	predict *factor* $\longrightarrow$ `id`
factor_tail term_tail stmt_list `$$`	`+ B write sum ...`	match `id`
term_tail stmt_list `$$`	`+ B write sum ...`	predict *factor_tail* $\longrightarrow \epsilon$
add_op term term_tail stmt_list `$$`	`+ B write sum ...`	predict *term_tail* $\longrightarrow$ *add_op term term_tail*
`+` *term term_tail stmt_list* `$$`	`+ B write sum ...`	predict *add_op* $\longrightarrow$ `+`
term term_tail stmt_list `$$`	`B write sum ...`	match `+`
factor factor_tail term_tail stmt_list `$$`	`B write sum ...`	predict *term* $\longrightarrow$ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`B write sum ...`	predict *factor* $\longrightarrow$ `id`
factor_tail term_tail stmt_list `$$`	`write sum ...`	match `id`
term_tail stmt_list `$$`	`write sum write ...`	predict *factor_tail* $\longrightarrow \epsilon$
stmt_list `$$`	`write sum write ...`	predict *term_tail* $\longrightarrow \epsilon$

stmt stmt_list `$$`	`write sum write` …	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
`write` *expr stmt_list* `$$`	`write sum write` …	predict *stmt* $\longrightarrow$ `write` *expr*
expr stmt_list `$$`	`sum write sum / 2`	match `write`
term term_tail stmt_list `$$`	`sum write sum / 2`	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`sum write sum / 2`	predict *term* $\longrightarrow$ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`sum write sum / 2`	predict *factor* $\longrightarrow$ `id`
factor_tail term_tail stmt_list `$$`	`write sum / 2`	match `id`
term_tail stmt_list `$$`	`write sum / 2`	predict *factor_tail* $\longrightarrow \epsilon$
stmt_list `$$`	`write sum / 2`	predict *term_tail* $\longrightarrow \epsilon$
stmt stmt_list `$$`	`write sum / 2`	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
`write` *expr stmt_list* `$$`	`write sum / 2`	predict *stmt* $\longrightarrow$ `write` *expr*
expr stmt_list `$$`	`sum / 2`	match `write`
term term_tail stmt_list `$$`	`sum / 2`	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`sum / 2`	predict *term* $\longrightarrow$ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`sum / 2`	predict *factor* $\longrightarrow$ `id`
factor_tail term_tail stmt_list `$$`	`/ 2`	match `id`
mult_op factor factor_tail term_tail stmt_list `$$`	`/ 2`	predict *factor_tail* $\longrightarrow$ *mult_op factor factor_tail*
`/` *factor factor_tail term_tail stmt_list* `$$`	`/ 2`	predict *mult_op* $\longrightarrow$ `/`
factor factor_tail term_tail stmt_list `$$`	`2`	match `/`
`number` *factor_tail term_tail stmt_list* `$$`	`2`	predict *factor* $\longrightarrow$ `number`
factor_tail term_tail stmt_list `$$`		match `number`
term_tail stmt_list `$$`		predict *factor_tail* $\longrightarrow \epsilon$
stmt_list `$$`		predict *term_tail* $\longrightarrow \epsilon$
`$$`		predict *stmt_list* $\longrightarrow \epsilon$

LL Parsing

- Parse tree for the average program (Figure 2.18)



LL Parsing

- LL(1) parse table for parsing for calculator language

Top-of-stack	Current input token										
nonterminal	id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

- Table-driven parsers are almost always constructed automatically by a parser generator.

LL Parsing

- The two most common obstacles to “LL(1)-ness” are
  - Left recursion
    - a non-terminal in a grammar can directly or indirectly produce a string that starts with itself
  - Common prefixes
    - two or more productions in a grammar have the same initial symbols

LL Parsing

- Problems trying to make a grammar LL(1)
  - left recursion
    - example:

```
id_list → id | id_list , id

              equivalently

id_list → id id_list_tail
id_list_tail → , id id_list_tail
             | epsilon
```

    - we can get rid of all left recursion mechanically in any grammar

LL Parsing

- Problems trying to make a grammar LL(1)
  - common prefixes: another thing that LL parsers can't handle
    - solved by "left-factoring”
    - example:

```
stmt → id := expr | id ( arg_list )

            equivalently

stmt → id id_stmt_tail
id_stmt_tail → := expr
             | ( arg_list)
```

    - we can eliminate common prefixes mechanically

LL Parsing and LR Parsing

- Consider the following grammar for a comma-separated list of identifiers, terminated by a semicolon:

```
id_list → id  id_list_tail
id_list_tail → , id  id_list_tail
id_list_tail → ;
```

- Parse tree for the string **A, B, C;**
  - Figure 2.14

LR(1) Grammar for the Previous Example

```
id_list → id_list_prefix ;
id_list_prefix → id_list_prefix , id
id_list_prefix → id
```

- Parse tree for the string **A, B, C;**
  - Figure 2.15

LR(1) Grammar for the Calculator Language

1. *program* ⟶ *stmt_list* `$$`
2. *stmt_list* ⟶ *stmt_list stmt*
3. *stmt_list* ⟶ *stmt*
4. *stmt* ⟶ `id := ` *expr*
5. *stmt* ⟶ `read id`
6. *stmt* ⟶ `write` *expr*
7. *expr* ⟶ *term*
8. *expr* ⟶ *expr add_op term*
9. *term* ⟶ *factor*
10. *term* ⟶ *term mult_op factor*
11. *factor* ⟶ `(` *expr* `)`
12. *factor* ⟶ `id`
13. *factor* ⟶ `number`
14. *add_op* ⟶ `+`
15. *add_op* ⟶ `-`
16. *mult_op* ⟶ `*`
17. *mult_op* ⟶ `/`

Parse tree: Figure 2.16