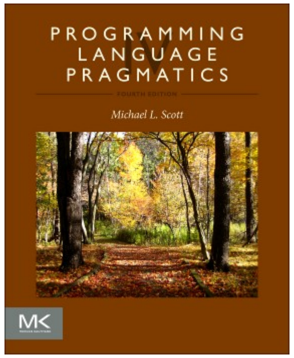
Chapter 2 :: Programming Language Syntax

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Regular Expressions

- A regular expression is one of the following:
  - A character
  - The empty string, denoted by ε
  - Two regular expressions concatenated
  - Two regular expressions separated by | (i.e., or)
  - A regular expression followed by the Kleene star * (concatenation of zero or more strings)

Regular Expressions

- Numerical constants accepted by a simple hand-held calculator:

$$
\begin{aligned}
&\mathit{number} &&\longrightarrow\ \ \mathit{integer} \mid \mathit{real}\\
&\mathit{integer} &&\longrightarrow\ \ \mathit{digit}\ \mathit{digit}^{*}\\
&\mathit{real} &&\longrightarrow\ \ \mathit{integer}\ \mathit{exponent} \mid \mathit{decimal}\ (\ \mathit{exponent} \mid \epsilon\ )\\
&\mathit{decimal} &&\longrightarrow\ \ \mathit{digit}^{*}\ (\ .\ \ \mathit{digit} \mid \mathit{digit}\ \ .\ )\ \mathit{digit}^{*}\\
&\mathit{exponent} &&\longrightarrow\ \ (\ \mathtt{e} \mid \mathtt{E}\ )\ (\ \mathtt{+} \mid \mathtt{-} \mid \epsilon\ )\ \mathit{integer}\\
&\mathit{digit} &&\longrightarrow\ \ \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}
\end{aligned}
$$

Context-Free Grammars

- The notation for context-free grammars (CFG) is sometimes called Backus-Naur Form (BNF)
- A CFG consists of
  - A set of *terminals* *T*
  - A set of *non-terminals* *N*
  - A *start symbol* *S* (a non-terminal)
  - A set of *productions*

ELSEVIER

Context-Free Grammars

- Expression grammar with precedence and associativity

$$
\begin{aligned}
\mathit{expr} &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ \mathit{expr} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \\
&\quad\ \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \\
\mathit{op} &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

Context-Free Grammars

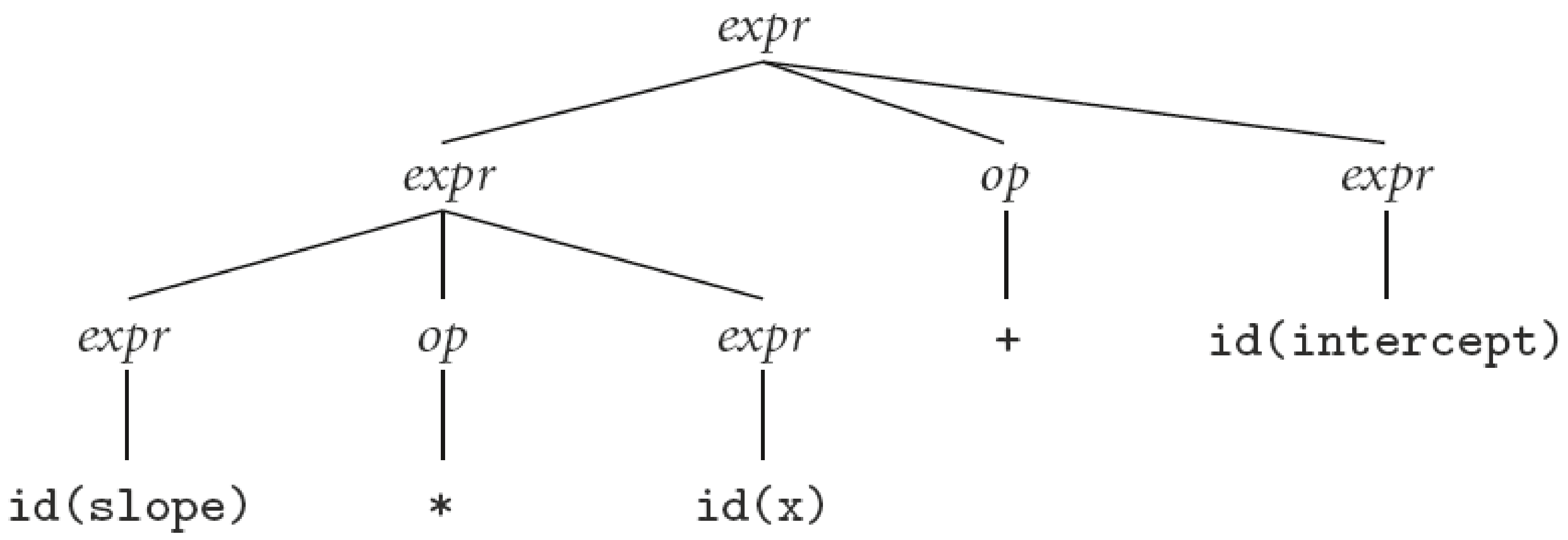
- In this grammar, generate the string **`"slope * x + intercept"`**

$$
\begin{aligned}
\mathit{expr} &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ \mathit{expr} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \\
&\qquad \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \\
\mathit{op} &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{expr} &\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underset{(\texttt{slope})}{\texttt{id}}\ \texttt{*}\ \underset{(\texttt{x})}{\texttt{id}}\ \texttt{+}\ \underset{(\texttt{intercept})}{\texttt{id}}
\end{aligned}
$$

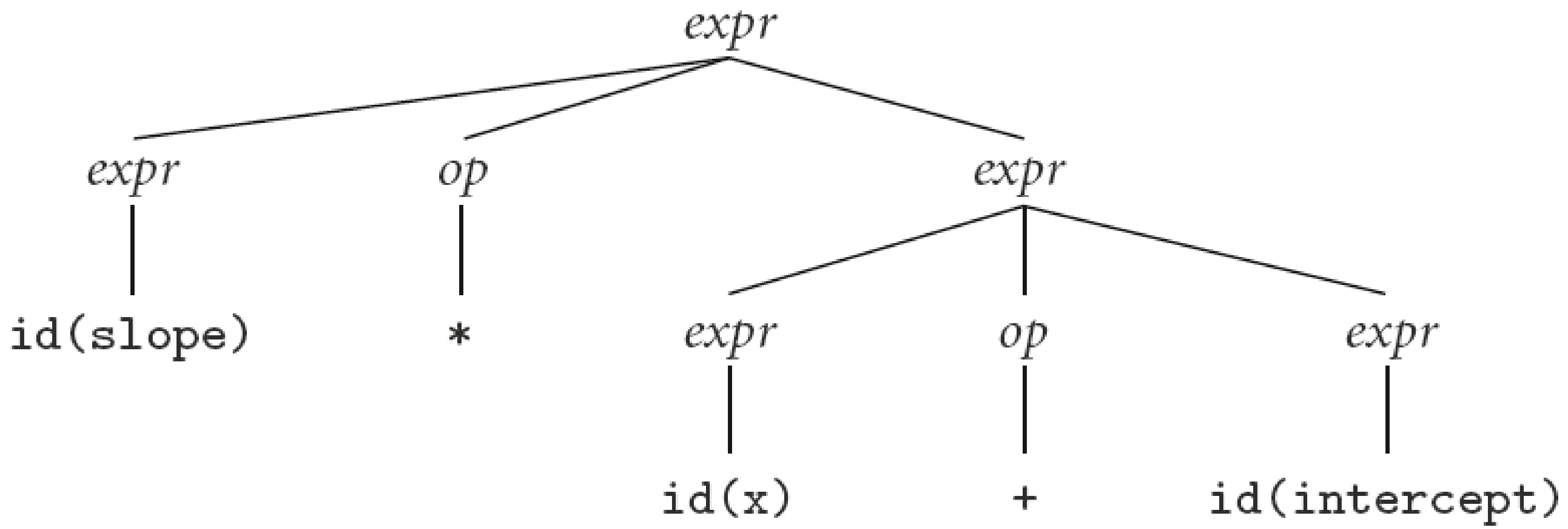
Context-Free Grammars

- Parse tree for expression grammar for **`"slope * x + intercept"`**



Context-Free Grammars

- Alternate (Incorrect) Parse tree for **"slope * x + intercept"**
- Our grammar is ambiguous



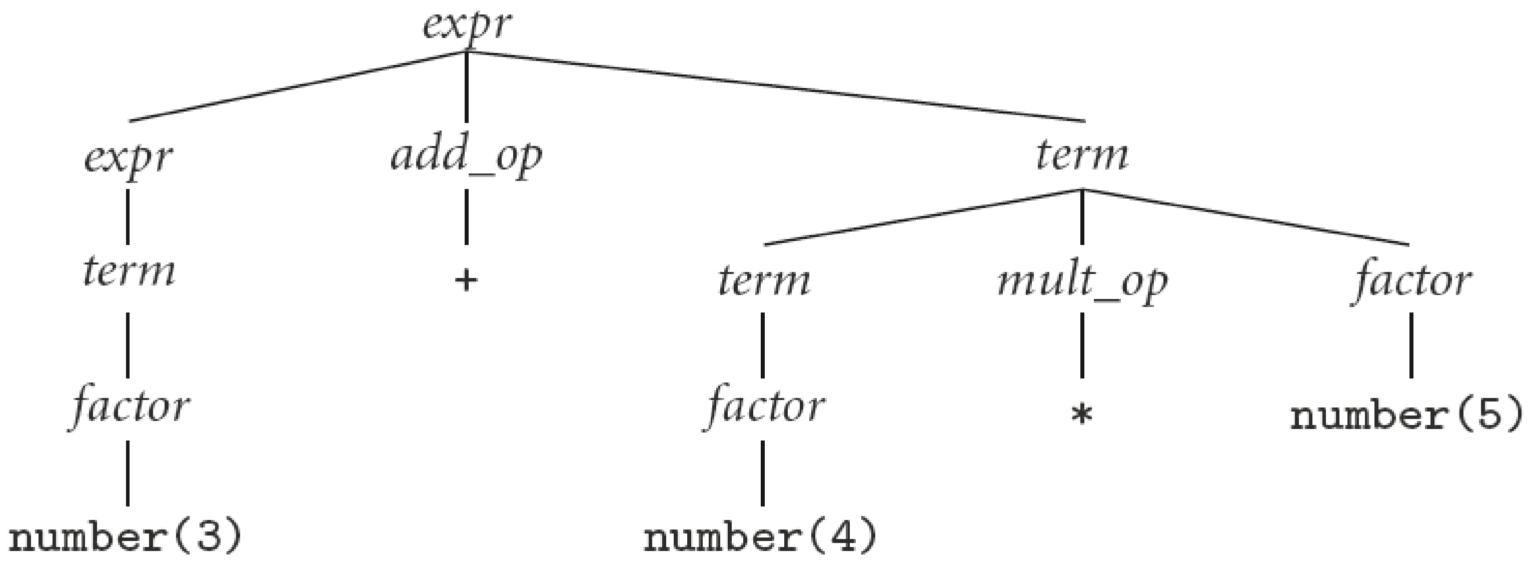
Context-Free Grammars

- A better version because it is unambiguous and captures precedence

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

Context-Free Grammars

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



Scanning

- Recall scanner is responsible for
  - tokenizing source
  - removing comments
  - (often) dealing with *pragmas* (i.e., significant comments)
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages

Scanning

- Suppose we are building an ad-hoc (hand-written) scanner for Pascal:
  - We read the characters one at a time with look-ahead
- If it is one of the one-character tokens
  `{ ( ) [ ] < > , ; = + - etc }`
  we announce that token
- If it is a ., we look at the next character
  - If that is a dot, we announce .
  - Otherwise, we announce . and reuse the look-ahead

Scanning

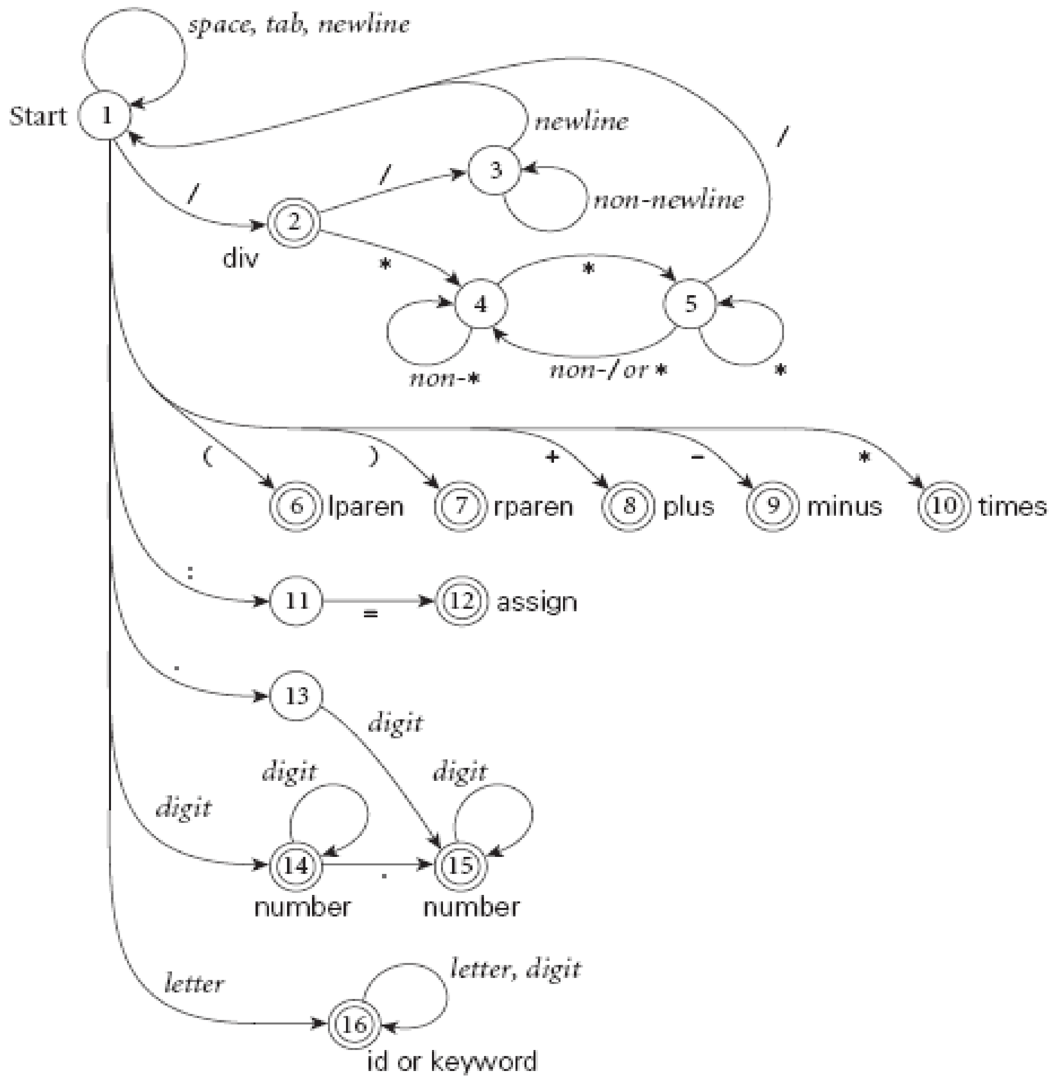
- If it is a `<`, we look at the next character
  - if that is a = we announce <=
  - otherwise, we announce < and reuse the look-ahead, etc
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore
  - then we check to see if it is a reserve word

Scanning

- If it is a digit, we keep reading until we find a non-digit
  - if that is not a . we announce an integer
  - otherwise, we keep looking for a real number
  - if the character after the . is not a digit we announce an integer and reuse the . and the look-ahead

Scanning

- Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton



Scanning

- This is a deterministic finite automaton (DFA)
  - Lex, scangen, etc. build these things automatically from a set of regular expressions
  - Specifically, they construct a machine that accepts the language

```
identifier | int const
| real const | comment | symbol
| ...
```

Scanning

- We run the machine over and over to get one token after another
  - Nearly universal rule:
    - always take the longest possible token from the input
      thus foobar is foobar and never f or foo or foob
    - more to the point, `3.14159` is a real const and never `3`, `.`, and `14159`
- Regular expressions "generate" a regular language; DFAs "recognize" it

Scanning

- Scanners tend to be built three ways
  - ad-hoc
  - semi-mechanical pure DFA
    (usually realized as nested case statements)
  - table-driven DFA
- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close

Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - though it's often easier to use perl, awk, sed
  - for details see Figure 2.11
- Table-driven DFA is what lex and scangen produce
  - lex (flex) in the form of C code
  - scangen in the form of numeric tables and a separate driver (for details see Figure 2.12)

Scanning

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
  - the next character will generally need to be saved for the next token
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
  - In Pascal, for example, when you have a 3 and you a see a dot
    - do you proceed (in hopes of getting 3.14)?
      or
    - do you stop (in fear of getting 3..5)?

Scanning

- In messier cases, you may not be able to get by with any fixed amount of look-ahead.In Fortr an, for example, we have

```
    DO 5 I = 1,25  loop
    DO 5 I = 1.25  assignment
```

- Here, we need to remember we were in a potentially final state, and save enough information that we can back up to it, if we get stuck later

Parsing

- Terminology:
  - context-free grammar (CFG)
  - symbols
    - terminals (tokens)
    - non-terminals
  - production
  - derivations (left-most and right-most - canonical)
  - parse trees
  - sentential form

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*
- There is an infinite number of grammars for every context-free language
  - not all grammars are created equal, however

Parsing

- It turns out that for any CFG we can create a parser that runs in O(n^3) time
- There are two well-known parsing algorithms that permit this
  - Early's algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- O(n^3) time is clearly unacceptable for a parser in a compiler - too slow

Parsing

- Fortunately, 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
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail on the differences between them

Parsing

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the *prefix property* (no valid string is a prefix of another valid string) has an LR(0) grammar

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
- The expression grammar (with precedence and associativity) you saw before is LR(1), but not LL(1)

LL Parsing

- Here is an LL(1) grammar (Fig 2.15):

```
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
•                    | ε
•     factor   →     ( expr )
•                    | id
•                    | number
•     add_op →       +
•                    | -
•     mult_op → *
•                    | /
```

LL Parsing

- Like the bottom-up grammar, this one captures associativity and precedence, but most people don't find it as pretty
  - for one thing, the operands of a given operator aren't in a RHS together!
  - however, the simplicity of the parsing algorithm makes up for this weakness
- How do we parse a string with this grammar?
  - by building the parse tree incrementally

LL Parsing

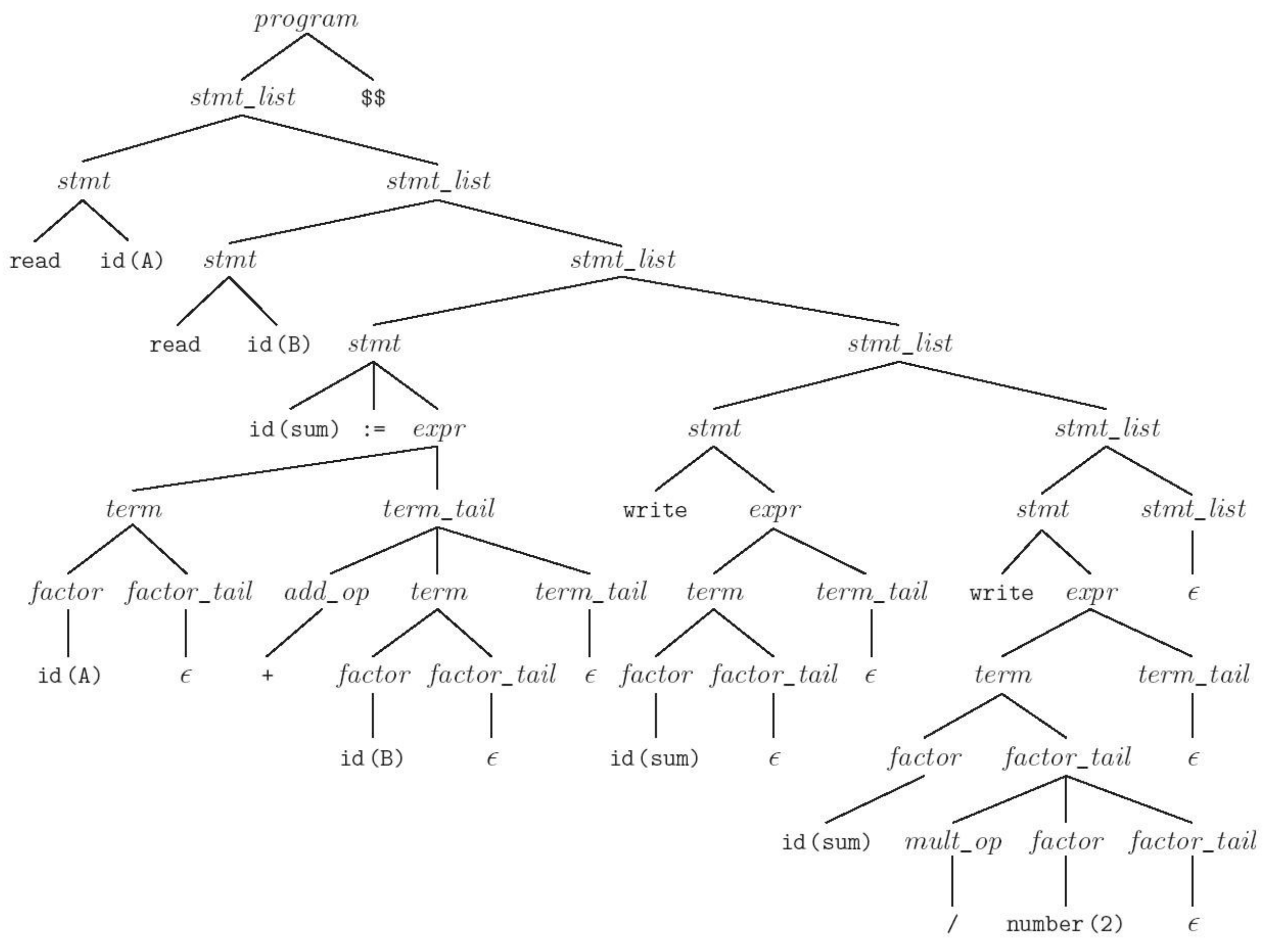
- Example (average program)

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

- We start at the top and predict needed productions on the basis of the current left-most non-terminal in the tree and the current input token

LL Parsing

- Parse tree for the average program (Figure 2.18)

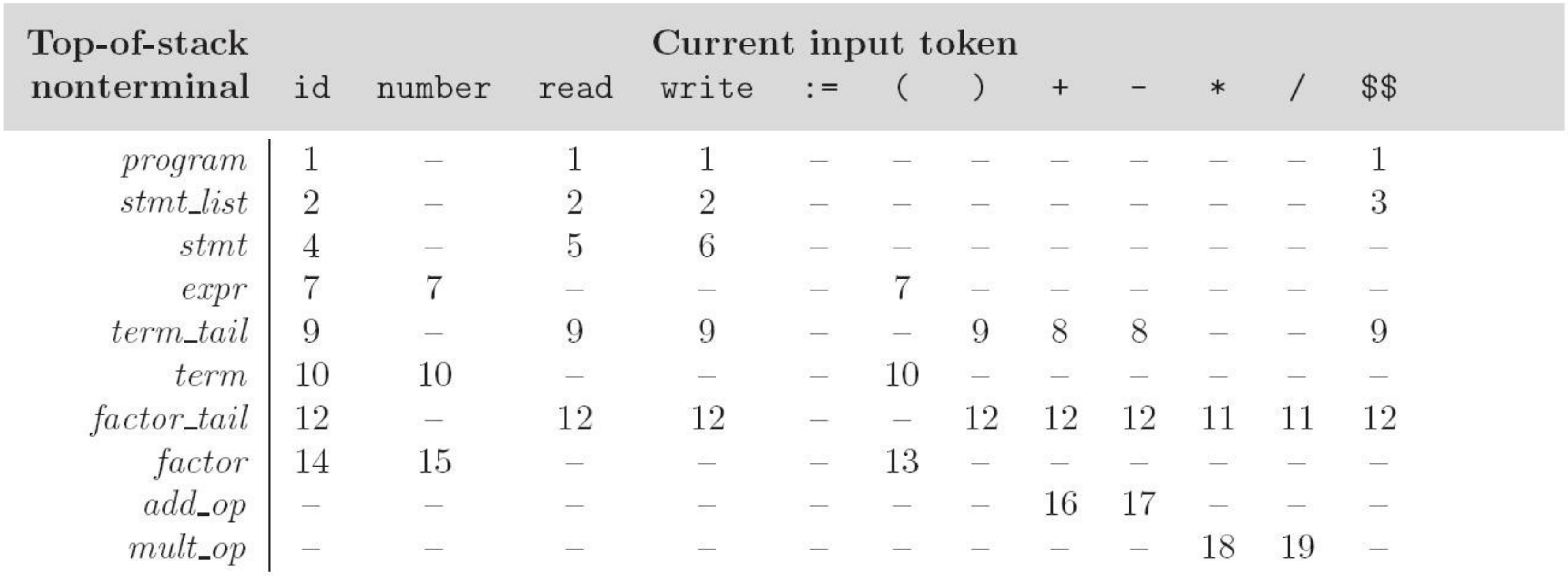


LL Parsing

- Table-driven LL parsing: you have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token. The actions are
  - (1) match a terminal
  - (2) predict a production
  - (3) announce a syntax error

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	–

LL Parsing

- To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack
  - for details see Figure 2.21
- The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program
  - what you *predict* you will see

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 left-factor mechanically

LL Parsing

- Note that eliminating left recursion and common prefixes does NOT make a grammar LL
  - there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine
  - the few that arise in practice, however, can generally be handled with kludges

LL Parsing

- Problems trying to make a grammar LL(1)
  - the"dangling else" problem prevents grammars from being LL(1) (or in fact LL(k) for any k)
  - the following natural grammar fragment is ambiguous (Pascal)

```
stmt → if cond then_clause else_clause
     | other_stuff
then_clause → then stmt
else_clause → else stmt
            | epsilon
```

ELSEVIER

LL Parsing

- The less natural grammar fragment can be parsed bottom-up but not top-down

```
stmt → balanced_stmt | unbalanced_stmt
balanced_stmt → if cond then balanced_stmt
                  else balanced_stmt
              | other_stuff
unbalanced_stmt → if cond then stmt
              | if cond then balanced_stmt
                  else     unbalanced_stmt
```

LL Parsing

- The usual approach, whether top-down OR bottom-up, is to use the ambiguous grammar together with a *disambiguating rule* that says
  - else goes with the closest then or
  - more generally, the first of two possible productions is the one to predict (or reduce)

LL Parsing

- Better yet, languages (since Pascal) generally employ explicit end-markers, which eliminate this problem
- In Modula-2, for example, one says:

```
if A = B then
      if C = D then E := F end
else
      G := H
end
```

- Ada says 'end if'; other languages say 'fi'

LL Parsing

- One problem with end markers is that they tend to bunch up. In Pascal you say

```
if A = B then …
else if A = C then …
else if A = D then …
else if A = E then …
else ...;
```

- With end markers this becomes

```
if A = B then …
else if A = C then …
else if A = D then …
else if A = E then …
else ...;
end; end; end; end;
```

LL Parsing

- The algorithm to build predict sets is tedious (for a "real" sized grammar), but relatively simple
- It consists of three stages:
  - (1) compute FIRST sets for symbols
  - (2) compute FOLLOW sets for non-terminals (this requires computing FIRST sets for some *strings*)
  - (3) compute predict sets or table for all productions

LL Parsing

- Algorithm First/Follow/Predict:
  - FIRST(α) == {a : α $\rightarrow^*$ a β} ∪ (if α =>* ε THEN {ε} ELSE NULL)
  - FOLLOW(A) == {a : S $\rightarrow^+$ α A a β} ∪ (if S →* α A THEN {ε} ELSE NULL)
  - Predict (A → X_1 ... X_m) == (FIRST (X_1 ... X_m) - {ε}) ∪ (if X_1, ..., X_m →* ε then FOLLOW (A) ELSE NULL)
- For examples, look at lecture notes.

LL Parsing

- If any token belongs to the predict set of more than one production with the same LHS, then the grammar is not LL(1)

LR Parsing

- LR parsers are almost always table-driven:
  - like a table-driven LL parser, an LR parser uses a big loop in which it repeatedly inspects a two-dimensional table to find out what action to take
  - unlike the LL parser, however, the LR driver has non-trivial state (like a DFA), and the table is indexed by current input token and current state
  - the stack contains a record of what has been seen SO FAR (NOT what is expected)

ELSEVIER

LR Parsing

- A scanner is a DFA
  - it can be specified with a state diagram
- An LL or LR parser is a PDA
  - Early's & CYK algorithms do NOT use PDAs
  - a PDA can be specified with a state diagram and a stack
    - the state diagram looks just like a DFA state diagram, except the arcs are labeled with <input symbol, top-of-stack symbol> pairs, and in addition to moving to a new state the PDA has the option of pushing or popping a finite number of symbols onto/off the stack

LR Parsing

- An SLR/LALR/LR PDA has multiple states
  - it is a "recognizer," not a "predictor"
  - it builds a parse tree from the bottom up
  - the states keep track of which productions we *might* be in the middle
- The parsing of the Characteristic Finite State Machine (CFSM) is based on
  - Shift
  - Reduce

LR Parsing

- This grammar is SLR(1), a particularly nice class of bottom-up grammar
  - it isn't exactly what we saw originally
  - we've eliminated the epsilon production to simplify the presentation
- For an example, look at Bottom-Up parsers lecture notes.

ELSEVIER