Chapter 2 ::
Programming Language Syntax

Programming Language Pragmatics

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 literals in Pascal may be generated by the following:

$$
\begin{aligned}
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{unsigned\_integer} &\longrightarrow \textit{digit}\ \textit{digit}\ * \\
\textit{unsigned\_number} &\longrightarrow \textit{unsigned\_integer}\ ((\ .\ \textit{unsigned\_integer}\ ) \mid \epsilon\ ) \\
&\quad (((\ \texttt{e} \mid \texttt{E}\ )\ (\ + \mid - \mid \epsilon\ )\ \textit{unsigned\_integer}\ ) \mid \epsilon\ )
\end{aligned}
$$

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*

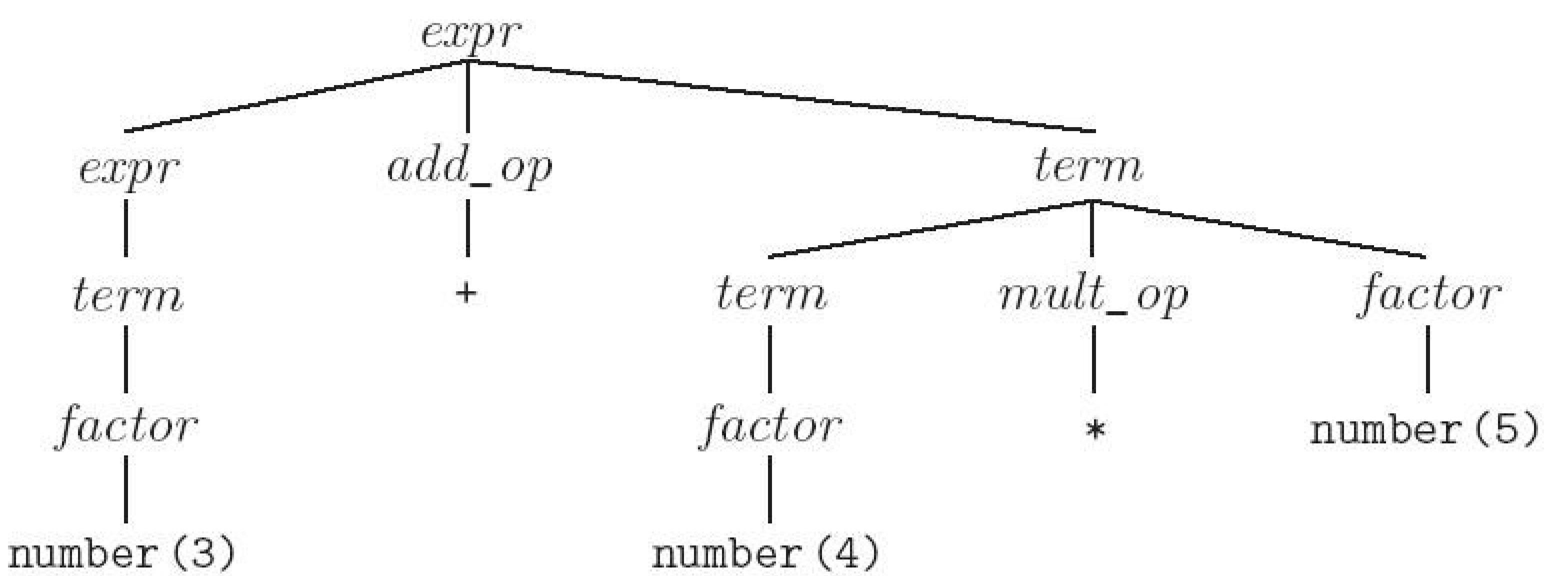
Context-Free Grammars

- Expression grammar with precedence and associativity

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. $mult\_op \longrightarrow \texttt{*} \mid \texttt{/}$

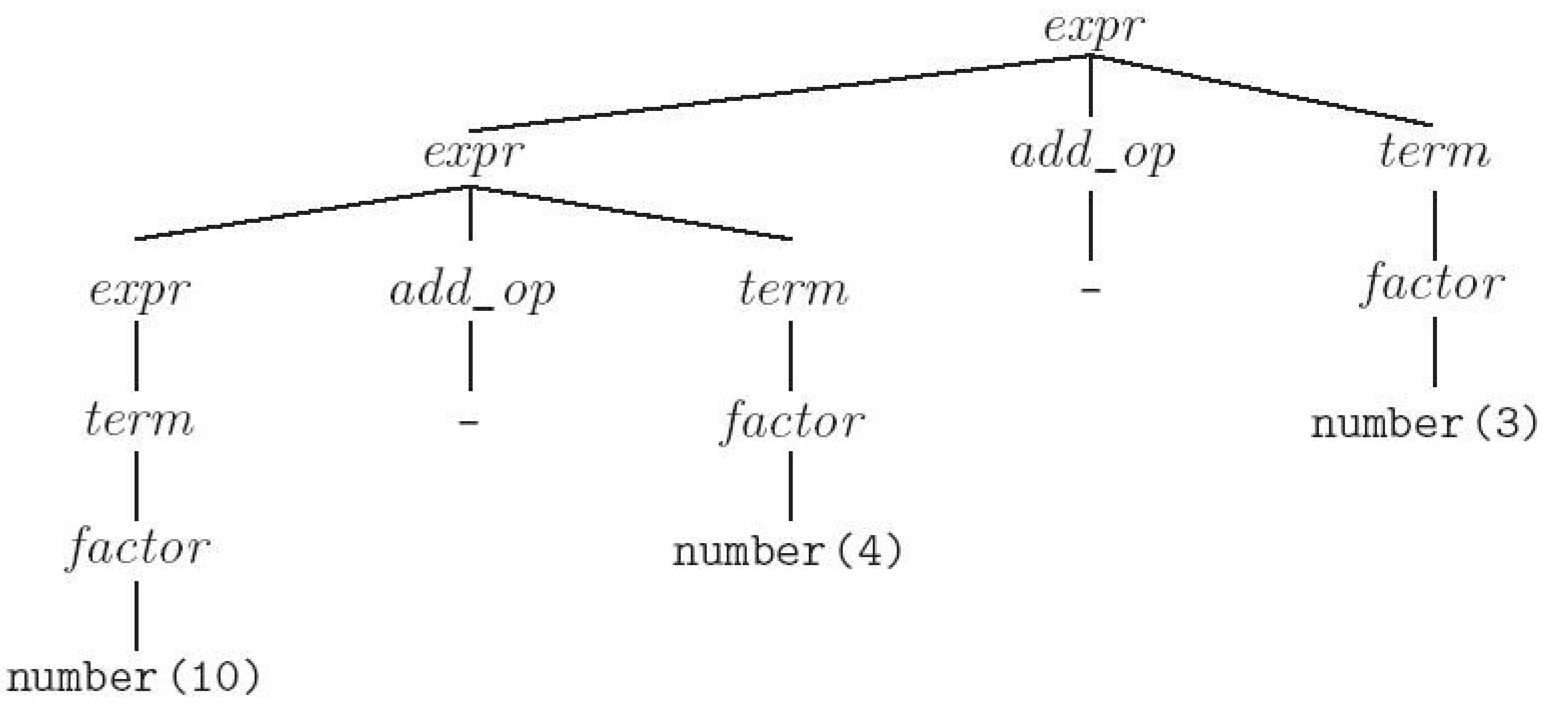
Context-Free Grammars

- 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`



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

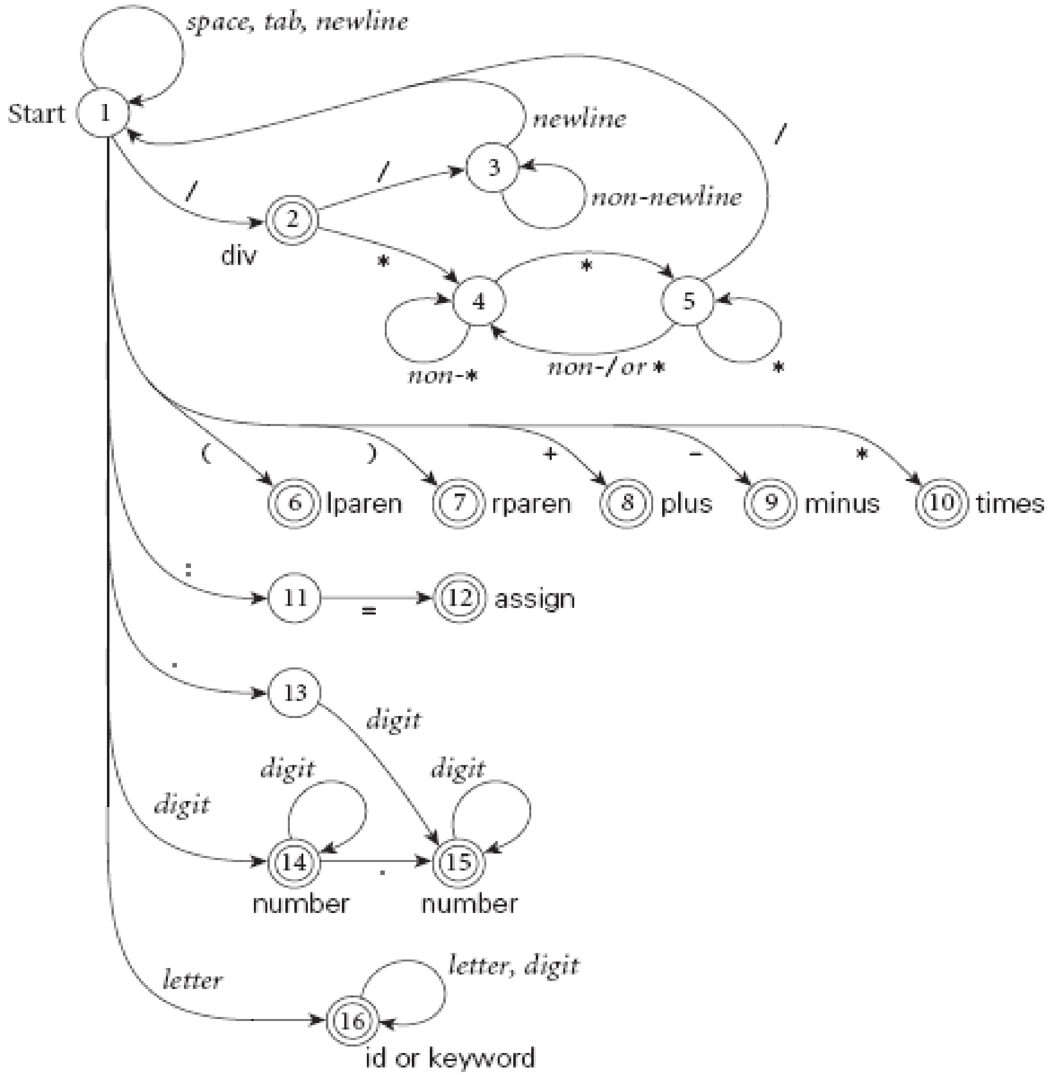
- If it is a `<`, we look at the next character
  - if that is a `=` we announce `<=`
  - otherwise, we announce `<` and reuse the look-ahead, etc
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore
  - then we check to see if it is a reserve word

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

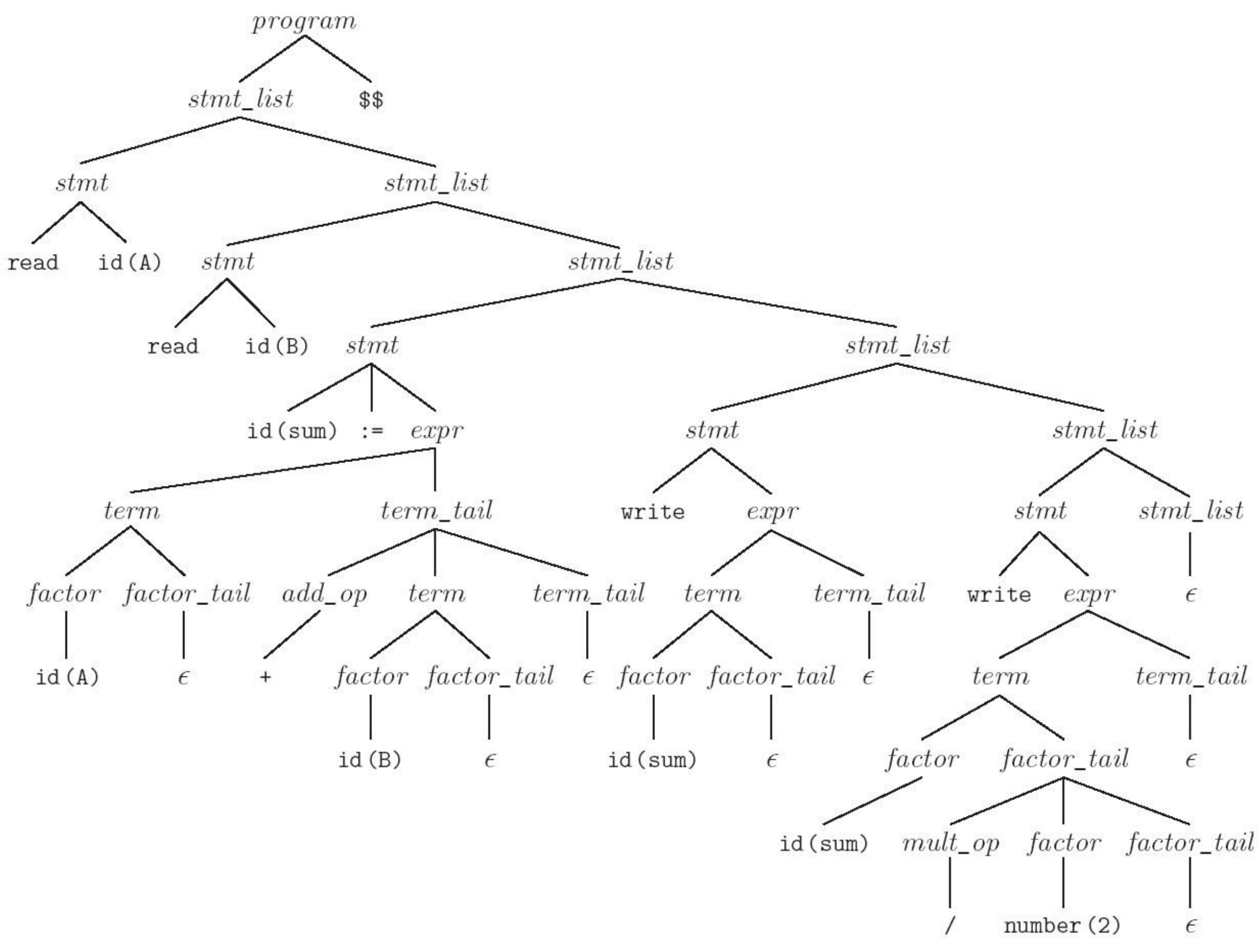
- Example (average program)

```
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 2.17)



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 nonterminal	Current input token												
	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 2.20
- 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
```

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

- It is conventional in general discussions of grammars to use
  - lower case letters near the beginning of the alphabet for terminals
  - lower case letters near the end of the alphabet for strings of terminals
  - upper case letters near the beginning of the alphabet for non-terminals
  - upper case letters near the end of the alphabet for arbitrary symbols
  - greek letters for arbitrary strings of symbols

LL Parsing

- Algorithm First/Follow/Predict:
  - $FIRST(\alpha) == \{a : \alpha \rightarrow^* a\ \beta\} \cup (\text{if } \alpha \Rightarrow^* \varepsilon \text{ THEN } \{\varepsilon\} \text{ ELSE NULL})$
  - $FOLLOW(A) == \{a : S \rightarrow^+ \alpha\ A\ a\ \beta\} \cup (\text{if } S \rightarrow^* \alpha\ A \text{ THEN } \{\varepsilon\} \text{ ELSE NULL})$
  - $Predict\ (A \rightarrow X_1 \ldots X_m) == (FIRST\ (X_1 \ldots X_m) - \{\varepsilon\}) \cup (\text{if } X_1, \ldots, X_m \rightarrow^* \varepsilon \text{ then } FOLLOW\ (A) \text{ ELSE NULL})$
- Details following…

LL Parsing

$program \longrightarrow stmt\_list$ `$$`	`$$` $\in$ FOLLOW($stmt\_list$), $\epsilon \in$ FOLLOW(`$$`), and $\epsilon \in$ FOLLOW($program$)
$stmt\_list \longrightarrow stmt\ stmt\_list$	
$stmt\_list \longrightarrow \epsilon$	$\epsilon \in$ FIRST($stmt\_list$)
$stmt \longrightarrow$ `id` `:=` $expr$	`id` $\in$ FIRST($stmt$) and `:=` $\in$ FOLLOW(`id`)
$stmt \longrightarrow$ `read` `id`	`read` $\in$ FIRST($stmt$) and `id` $\in$ FOLLOW(`read`)
$stmt \longrightarrow$ `write` $expr$	`write` $\in$ FIRST($stmt$)
$expr \longrightarrow term\ term\_tail$	
$term\_tail \longrightarrow add\_op\ term\ term\_tail$	
$term\_tail \longrightarrow \epsilon$	$\epsilon \in$ FIRST($term\_tail$)
$term \longrightarrow factor\ factor\_tail$	
$factor\_tail \longrightarrow mult\_op\ factor\ factor\_tail$	
$factor\_tail \longrightarrow \epsilon$	$\epsilon \in$ FIRST($factor\_tail$)
$factor \longrightarrow$ `(` $expr$ `)`	`(` $\in$ FIRST($factor$) and `)` $\in$ FOLLOW($expr$)
$factor \longrightarrow$ `id`	`id` $\in$ FIRST($factor$)
$factor \longrightarrow$ `number`	`number` $\in$ FIRST($factor$)
$add\_op \longrightarrow$ `+`	`+` $\in$ FIRST(add_op)
$add\_op \longrightarrow$ `-`	`-` $\in$ FIRST(add_op)
$mult\_op \longrightarrow$ `*`	`*` $\in$ FIRST($mult\_op$)
$mult\_op \longrightarrow$ `/`	`/` $\in$ FIRST($mult\_op$)

Figure 2.21: **"Obvious" facts about the LL(1) calculator grammar.**

LL Parsing

FIRST

program {id, read, write, $$}
stmt_list {id, read, write, ϵ}
stmt {id, read, write}
expr {(, id, number}
term_tail {+, -, ϵ}
term {(, id, number}
factor_tail {*, /, ϵ}
factor {(, id, number}
add_op {+, -}
mult_op {*, /}

Also note that FIRST(a) = {a} $\forall$ tokens a.

FOLLOW

id {+, -, *, /,), :=, id, read, write, $$}
number {+, -, *, /,), id, read, write, $$}
read {id}
write {(, id, number}
({(, id, number}
) {+, -, *, /,), id, read, write, $$}
:= {(, id, number}
+ {(, id, number}
- {(, id, number}
* {(, id, number}
/ {(, id, number}
$$ {ϵ}
program {ϵ}
stmt_list {$$}
stmt {id, read, write, $$}
expr {), id, read, write, $$}
term_tail {), id, read, write, $$}
term {+, -,), id, read, write, $$}
factor_tail {+, -,), id, read, write, $$}
factor {+, -, *, /,), id, read, write, $$}
add_op {(, id, number}
mult_op {(, id, number}

PREDICT

1 *program* $\longrightarrow$ *stmt_list* $$ {id, read, write, $$}
2 *stmt_list* $\longrightarrow$ *stmt stmt_list* {id, read, write}
3 *stmt_list* $\longrightarrow$ ϵ {$$}
4 *stmt* $\longrightarrow$ id := *expr* {id}
5 *stmt* $\longrightarrow$ read id {read}
6 *stmt* $\longrightarrow$ write *expr* {write}
7 *expr* $\longrightarrow$ *term term_tail* {(, id, number}
8 *term_tail* $\longrightarrow$ *add_op term term_tail* {+, -}
9 *term_tail* $\longrightarrow$ ϵ {), id, read, write, $$}
10 *term* $\longrightarrow$ *factor factor_tail* {(, id, number}
11 *factor_tail* $\longrightarrow$ *mult_op factor factor_tail* {*, /}
12 *factor_tail* $\longrightarrow$ ϵ {+, -,), id, read, write, $$}
13 *factor* $\longrightarrow$ (*expr*) {(}
14 *factor* $\longrightarrow$ id {id}
15 *factor* $\longrightarrow$ number {number}
16 *add_op* $\longrightarrow$ + {+}
17 *add_op* $\longrightarrow$ - {-}
18 *mult_op* $\longrightarrow$ * {*}
19 *mult_op* $\longrightarrow$ / {/}

Figure 2.22: FIRST, FOLLOW, and PREDICT sets for the calculator language.

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)
- A conflict can arise because
  - the same token can begin more than one RHS
  - it can begin one RHS and can also appear *after* the LHS in some valid program, and one possible RHS is ε

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)

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 LL(1) PDA has only one state!
  - well, actually two; it needs a second one to accept with, but that's all (it's pretty simple)
  - all the arcs are self loops; the only difference between them is the choice of whether to push or pop
  - the final state is reached by a transition that sees EOF on the input and 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

- To illustrate LR parsing, consider the grammar (Figure 2.24, Page 73):

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

LR Parsing

- LR grammar (continued):

```
9. term   →    factor
10.            | term mult_op factor
11.  factor →( expr )
12.            | id
13.            | number
14.  add op → +
15.            | -
16.  mult op → *
17.            | /
```


ELSEVIER

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 details on the table driven SLR(1) parsing please note the following slides

LR Parsing

State	Transitions
0. *program* → • *stmt_list* `$$`	on *stmt_list* shift and goto 2
stmt_list → • *stmt_list stmt* *stmt_list* → • *stmt* *stmt* → • `id := ` *expr* *stmt* → • `read id` *stmt* → • `write` *expr*	 on *stmt* shift and reduce (pop 1 state, push *stmt_list* on inpu on `id` shift and goto 3 on `read` shift and goto 1 on `write` shift and goto 4
1. *stmt* → `read` • `id`	on `id` shift and reduce (pop 2 states, push *stmt* on input)
2. *program* → *stmt_list* • `$$` *stmt_list* → *stmt_list* • *stmt*	on `$$` shift and reduce (pop 2 states, push *program* on input) on *stmt* shift and reduce (pop 2 states, push *stmt_list* on inpu
stmt → • `id := ` *expr* *stmt* → • `read id` *stmt* → • `write` *expr*	on `id` shift and goto 3 on `read` shift and goto 1 on `write` shift and goto 4
3. *stmt* → `id` • `:=` *expr*	on `:=` shift and goto 5
4. *stmt* → `write` • *expr*	on *expr* shift and goto 6
expr → • *term* *expr* → • *expr add_op term* *term* → • *factor* *term* → • *term mult_op factor* *factor* → • `(` *expr* `)` *factor* → • `id` *factor* → • `number`	on *term* shift and goto 7 on *factor* shift and reduce (pop 1 state, push *term* on input) on `(` shift and goto 8 on `id` shift and reduce (pop 1 state, push *factor* on input) on `number` shift and reduce (pop 1 state, push *factor* on inpu
5. *stmt* → `id :=` • *expr*	on *expr* shift and goto 9
expr → • *term* *expr* → • *expr add_op term* *term* → • *factor* *term* → • *term mult_op factor* *factor* → • `(` *expr* `)` *factor* → • `id` *factor* → • `number`	on *term* shift and goto 7 on *factor* shift and reduce (pop 1 state, push *term* on input) on `(` shift and goto 8 on `id` shift and reduce (pop 1 state, push *factor* on input) on `number` shift and reduce (pop 1 state, push *factor* on inpu
6. *stmt* → `write` *expr* • *stmt* → *expr* • *add_op term*	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce (pop 2 states, push *stmt* on input) on *add_op* shift and goto 10
add_op → • `+` *add_op* → • `-`	on `+` shift and reduce (pop 1 state, push *add_op* on input) on `-` shift and reduce (pop 1 state, push *add_op* on input)

Figure 2.25: **CFSM for the calculator grammar (Figure 2.24).** Basis and closure items in each state are separated by a horizontal rule. Trivial reduce-only states have been eliminated by use of "shift and reduce" transitions *(continued)*.

State	Transitions
7. *expr* → *term* • *term* → *term* • *mult_op factor*	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce (pop 1 state, push *expr* on input) on *mult_op* shift and goto 11
mult_op → • `*` *mult_op* → • `/`	on `*` shift and reduce (pop 1 state, push *mult_op* on input) on `/` shift and reduce (pop 1 state, push *mult_op* on input)
8. *factor* → `(` • *expr* `)`	on *expr* shift and goto 12
expr → • *term* *expr* → • *expr add_op term* *term* → • *factor* *term* → • *term mult_op factor* *factor* → • `(` *expr* `)` *factor* → • `id` *factor* → • `number`	on *term* shift and goto 7 on *factor* shift and reduce (pop 1 state, push *term* on input) on `(` shift and goto 8 on `id` shift and reduce (pop 1 state, push *factor* on input) on `number` shift and reduce (pop 1 state, push *factor* on input)
9. *stmt* → `id :=` *expr* • *expr* → *expr* • *add_op term*	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce (pop 3 states, push *stmt* on input) on *add_op* shift and goto 10
add_op → • `+` *add_op* → • `-`	on `+` shift and reduce (pop 1 state, push *add_op* on input) on `-` shift and reduce (pop 1 state, push *add_op* on input)
10. *expr* → *expr add_op* • *term*	on *term* shift and goto 13
term → • *factor* *term* → • *term mult_op factor* *factor* → • `(` *expr* `)` *factor* → • `id` *factor* → • `number`	on *factor* shift and reduce (pop 1 state, push *term* on input) on `(` shift and goto 8 on `id` shift and reduce (pop 1 state, push *factor* on input) on `number` shift and reduce (pop 1 state, push *factor* on input)
11. *term* → *term mult_op* • *factor*	on *factor* shift and reduce (pop 3 states, push *term* on input)
factor → • `(` *expr* `)` *factor* → • `id` *factor* → • `number`	on `(` shift and goto 8 on `id` shift and reduce (pop 1 state, push *factor* on input) on `number` shift and reduce (pop 1 state, push *factor* on input)
12. *factor* → `(` *expr* • `)` *expr* → *expr* • *add_op term*	on `)` shift and reduce (pop 3 states, push *factor* on input) on *add_op* shift and goto 10
add_op → • `+` *add_op* → • `-`	on `+` shift and reduce (pop 1 state, push *add_op* on input) on `-` shift and reduce (pop 1 state, push *add_op* on input)
13. *expr* → *expr add_op term* • *term* → *term* • *mult_op factor*	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce (pop 3 states, push *expr* on input) on *mult_op* shift and goto 11
mult_op → • `*` *mult_op* → • `/`	on `*` shift and reduce (pop 1 state, push *mult_op* on input) on `/` shift and reduce (pop 1 state, push *mult_op* on input)

Figure 2.25: *(continued)*

LR Parsing

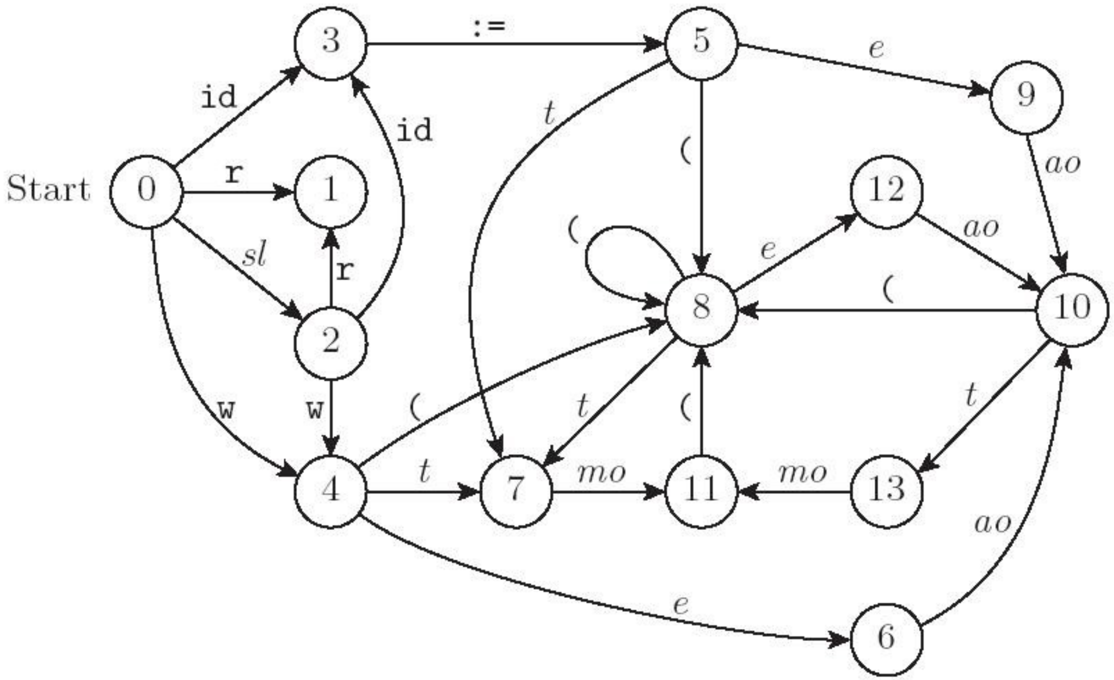


Figure 2.26: **Pictorial representation of the CFSM of Figure 2.25.** Symbol names have been abbreviated for clarity. Reduce actions are not shown.

LR Parsing

Top-of-stack	Current input symbol																		
state	*sl*	*s*	*e*	*t*	*f*	*ao*	*mo*	`id`	`lit`	`r`	`w`	`:=`	`(`	`)`	`+`	`-`	`*`	`/`	`$$`
0	s2	b3	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	b5	–	–	–	–	–	–	–	–	–	–	–
2	–	b2	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	–	–	–	–	–	–	–	–	s5	–	–	–	–	–	–	–
4	–	–	s6	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
5	–	–	s9	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
6	–	–	–	–	–	s10	–	r6	–	r6	r6	–	–	–	b14	b15	–	–	r6
7	–	–	–	–	–	–	s11	r7	–	r7	r7	–	–	r7	r7	r7	b16	b17	r7
8	–	–	s12	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
9	–	–	–	–	–	s10	–	r4	–	r4	r4	–	–	–	b14	b15	–	–	r4
10	–	–	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	–	–	–	–	b11	b14	b15	–	–	–
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	r8

Figure 2.27: **SLR(1) parse table for the calculator language.** Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.24. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand side symbol and right-hand side length for each production.

LR Parsing

- SLR parsing is based on
  - Shift
  - Reduce

  and also
  - Shift & Reduce (for optimization)

Parse stack	Input stream	Comment
0	read A read B ...	
0 read 1	A read B ...	shift read
0	stmt read B ...	shift id(A) & reduce by stmt → read id
0	stmt_list read B ...	shift stmt & reduce by stmt_list → stmt
0 stmt_list 2	read B sum ...	shift stmt_list
0 stmt_list 2 read 1	B sum := ...	shift read
0 stmt_list 2	stmt sum := ...	shift id(B) & reduce by stmt → read id
0	stmt_list sum := ...	shift stmt & reduce by stmt_list → stmt_list stmt
0 stmt_list 2	sum := A ...	shift stmt_list
0 stmt_list 2 id 3	:= A + ...	shift id(sum)
0 stmt_list 2 id 3 := 5	A + B ...	shift :=
0 stmt_list 2 id 3 := 5	factor + B ...	shift id(A) & reduce by factor → id
0 stmt_list 2 id 3 := 5	term + B ...	shift factor & reduce by term → factor
0 stmt_list 2 id 3 := 5 term 7	+ B write ...	shift term
0 stmt_list 2 id 3 := 5	expr + B write ...	reduce by expr → term
0 stmt_list 2 id 3 := 5 expr 9	+ B write ...	shift expr
0 stmt_list 2 id 3 := 5 expr 9	add_op B write ...	shift + & reduce by add_op → +
0 stmt_list 2 id 3 := 5 expr 9 add_op 10	B write sum ...	shift add_op
0 stmt_list 2 id 3 := 5 expr 9 add_op 10	factor write sum ...	shift id(B) & reduce by factor → id
0 stmt_list 2 id 3 := 5 expr 9 add_op 10	term write sum ...	shift factor & reduce by term → factor
0 stmt_list 2 id 3 := 5 expr 9 add_op 10 term 13	write sum ...	shift term
0 stmt_list 2 id 3 := 5	expr write sum ...	reduce by expr → expr add_op term
0 stmt_list 2 id 3 := 5 expr 9	write sum ...	shift expr
0 stmt_list 2	stmt write sum ...	reduce by stmt → id := expr
0	stmt_list write sum ...	shift stmt & reduce by stmt_list → stmt
0 stmt_list 2	write sum ...	shift stmt_list
0 stmt_list 2 write 4	sum write sum ...	shift write
0 stmt_list 2 write 4	factor write sum ...	shift id(sum) & reduce by factor → id
0 stmt_list 2 write 4	term write sum ...	shift factor & reduce by term → factor
0 stmt_list 2 write 4 term 7	write sum ...	shift term
0 stmt_list 2 write 4	expr write sum ...	reduce by expr → term
0 stmt_list 2 write 4 expr 6	write sum ...	shift expr
0 stmt_list 2	stmt write sum ...	reduce by stmt → write expr
0	stmt_list write sum ...	shift stmt & reduce by stmt_list → stmt_list stmt
0 stmt_list 2	write sum / ...	shift stmt_list
0 stmt_list 2 write 4	sum / 2 ...	shift write
0 stmt_list 2 write 4	factor / 2 ...	shift id(sum) & reduce by factor → id
0 stmt_list 2 write 4	term / 2 ...	shift factor & reduce by term → factor
0 stmt_list 2 write 4 term 7	/ 2 $$	shift term
0 stmt_list 2 write 4 term 7	mult_op 2 $$	shift / & reduce by mult_op → /
0 stmt_list 2 write 4 term 7 mult_op 11	2 $$	shift mult_op
0 stmt_list 2 write 4 term 7 mult_op 11	factor $$	shift number(2) & reduce by factor → number
0 stmt_list 2 write 4	term $$	shift factor & reduce by term → term mult_op factor
0 stmt_list 2 write 4 term 7	$$	shift term
0 stmt_list 2 write 4	expr $$	reduce by expr → term
0 stmt_list 2 write 4 expr 6	$$	shift expr
0 stmt_list 2	stmt $$	reduce by stmt → write expr
0	stmt_list $$	shift stmt & reduce by stmt_list → stmt_list stmt
0 stmt_list 2	$$	shift stmt_list
0	program	shift $$ & reduce by program → stmt_list $$
[done]		

Figure 2.29: **Trace of a table-driven SLR(1) parse of the sum-and-average program.** States in the parse stack are shown in boldface type. Symbols in the parse stack are for clarity only; they are not needed by the parsing algorithm. Parsing begins with the initial state of the CFSM (State 0) in the stack. It ends when we reduce by *program* → *stmt_list* $$, uncovering State 0 again and pushing *program* onto the input stream.