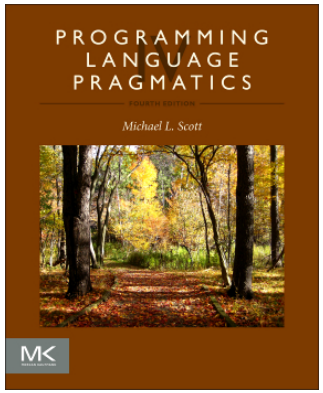
Parsing Teaser

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 2 through section 2.2

ELSEVIER

Parsing

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

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

Parsing

- You will see LL(n) or LR(n)
  - This number indicates how many tokens of look-ahead are required in order to parse
  - Almost all real compilers use 1 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 factor fact_tail
12.                 | ε
13.  factor   →     ( expr )
14.                 | id
15.                 | number
16.  add_op →       +
17.                 | -
18.  mult_op →      *
19.                 | /
```

LL Parsing

- Table-driven LL parsing: a big loop which repeatedly looks 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, 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
```

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 push-down automaton (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

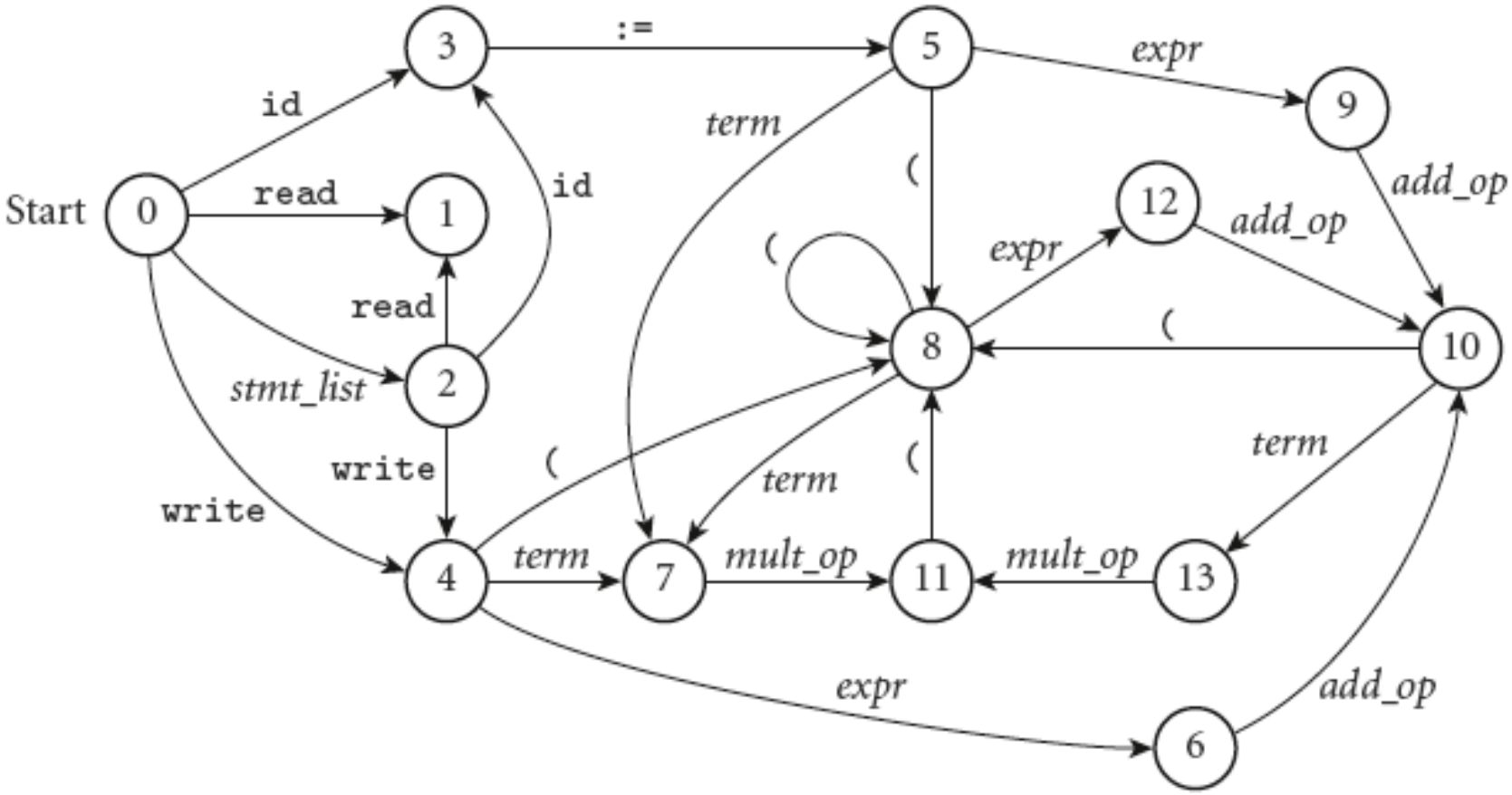


Figure 2.27 Pictorial representation of the CFSM of Figure 2.26. Reduce actions are not shown.

LR Parsing

Top-of-stack state	Current input symbol																		
	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 2.28 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 2.25. 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.