FIRST

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

Also note that FIRST(a) = {a} ∀ tokens a.

FOLLOW

id {+, -, *, /,), :=, id, read, write, $$}
literal {+, -, *, /,), id, read, write, $$}
read {id}
write {(, id, literal}
({(, id, literal}
) {+, -, *, /,), id, read, write, $$}
:= {(, id, literal}
+ {(, id, literal}
- {(, id, literal}
* {(, id, literal}
/ {(, id, literal}
$$ {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, literal}
mult_op {(, id, literal}

PREDICT

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

Figure 2.16 FIRST, FOLLOW, **and** PREDICT **sets for the calculator language.**

more serious version of the immediate error detection problem described in Section 2.2.4. There we saw that the use of context-independent FOLLOW sets could cause us to overlook a syntax error until after we had needlessly predicted one or more epsilon productions. Context-specific FOLLOW sets solved the problem, but did not change the set of *valid* programs that could be parsed with one token of look-ahead. If we define LL(k) to be the set of all grammars that can be parsed predictively using the top-of-stack symbol and k tokens of look-ahead, then it turns out that for $k > 1$ we must adopt a context-specific notion of FOLLOW sets in order to parse correctly. Our previous parsing algorithm—the one based on context-independent FOLLOW sets—is really SLL, rather than true LL. For $k = 1$, the two algorithms can parse the same set of grammars. For $k > 1$, LL is strictly more powerful.

Top-of-stack nonterminal	Current input token `id`	`literal`	`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	–

Figure 2.17 LL(1) parse table for the calculator language. Table entries indicate the production to predict (as numbered in Figure 2.14). A dash indicates an error. When the top-of-stack symbol is a terminal, the appropriate action is always to match it against an incoming token from the scanner.

Writing an LL(1) Grammar

When working with a top-down parser generator, one has to acquire a certain facility in writing and modifying LL(1) grammars. The two most common obstacles to "LL(1)-ness" are *left recursion* and *common prefixes.*

Left recursion occurs when the first symbol on the right-hand side of a production is the same as the symbol on the left-hand side. Here again is the grammar from page 51, which cannot be parsed top-down:

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

The problem is in the second and third productions; with *id_list_prefix* at top-of-stack and an `id` on the input, a predictive parser cannot tell which of the productions it should use. (Recall that left recursion is *desirable* in bottom-up grammars, because it allows recursive constructs to be discovered incrementally, as in Figure 2.9.)

Common prefixes occur when two different productions with the same left-hand side begin with the same symbol or symbols. Here is an example that commonly appears in Algol-family languages:

stmt → `id` `:=` *expr*
→ `id` `(` *argument_list* `)` —— procedure call

Clearly `id` is in the FIRST set of both right-hand sides, and therefore in the PREDICT set of both productions.

Read A
Read B
sum := A + B
write sum
write sum/2

Parse stack	Input stream	Comment
program	read A read B ...	initial stack contents
stmt_list $$	read A read B ...	predict program → stmt_list $$
stmt stmt_list $$	read A read B ...	predict stmt_list → stmt stmt_list
read id stmt_list $$	read A read B ...	predict stmt → read id
id stmt_list $$	A read B ...	match read
stmt_list $$	read B sum := ...	match id
stmt stmt_list $$	read B sum := ...	predict stmt_list → stmt stmt_list
read id stmt_list $$	read B sum := ...	predict stmt → read id
id stmt_list $$	B sum := ...	match read
stmt_list $$	sum := A + B ...	match id
stmt stmt_list $$	sum := A + B ...	predict stmt_list → stmt stmt_list
id := expr stmt_list $$	sum := A + B ...	predict stmt → id := expr
:= expr stmt_list $$	:= A + B ...	match id
expr stmt_list $$	A + B ...	match :=
term term_tail stmt_list $$	A + B ...	predict expr → term term_tail
factor factor_tail term_tail stmt_list $$	A + B ...	predict term → factor factor_tail
id factor_tail term_tail stmt_list $$	A + B ...	predict factor → id
factor_tail term_tail stmt_list $$	+ B write sum ...	match id
term_tail stmt_list $$	+ B write sum ...	predict factor_tail → ϵ
add_op term term_tail stmt_list $$	+ B write sum ...	predict term_tail → add_op term term_tail
+ term term_tail stmt_list $$	+ B write sum ...	predict add_op → +
term term_tail stmt_list $$	B write sum ...	match +
factor factor_tail term_tail stmt_list $$	B write sum ...	predict term → factor factor_tail
id factor_tail term_tail stmt_list $$	B write sum ...	predict factor → id
factor_tail term_tail stmt_list $$	write sum ...	match id
term_tail stmt_list $$	write sum write ...	predict factor_tail → ϵ
stmt_list $$	write sum write ...	predict term_tail → ϵ
stmt stmt_list $$	write sum write ...	predict stmt_list → stmt stmt_list
write expr stmt_list $$	write sum write ...	predict stmt → write expr
expr stmt_list $$	sum write sum / 2	match write
term term_tail stmt_list $$	sum write sum / 2	predict expr → term term_tail
factor factor_tail term_tail stmt_list $$	sum write sum / 2	predict term → factor factor_tail
id factor_tail term_tail stmt_list $$	sum write sum / 2	predict factor → id
factor_tail term_tail stmt_list $$	sum write sum / 2	match id
term_tail stmt_list $$	write sum / 2	predict factor_tail → ϵ
stmt_list $$	write sum / 2	predict term_tail → ϵ
stmt stmt_list $$	write sum / 2	predict stmt_list → stmt stmt_list
write expr stmt_list $$	write sum / 2	predict stmt → write expr
expr stmt_list $$	sum / 2	match write
term term_tail stmt_list $$	sum / 2	predict expr → term term_tail
factor factor_tail term_tail stmt_list $$	sum / 2	predict term → factor factor_tail
id factor_tail term_tail stmt_list $$	sum / 2	predict factor → id
factor_tail term_tail stmt_list $$	/ 2	match id
mult_op factor factor_tail term_tail stmt_list $$	/ 2	predict factor_tail → mult_op factor factor_tail
/ factor factor_tail term_tail stmt_list $$	/ 2	predict mult_op → /
factor factor_tail term_tail stmt_list $$	2	match /
literal factor_tail term_tail stmt_list $$	2	predict factor → literal
factor_tail term_tail stmt_list $$		match literal
term_tail stmt_list $$		predict factor_tail → ϵ
stmt_list $$		predict term_tail → ϵ
$$		predict stmt_list → ϵ

Figure 2.13 Trace of a table-driven LL(1) parse of the sum-and-average program.