3

Parsing

syn-tax: the way in which words are put together to form phrases, clauses, or sentences.

Webster's Dictionary

The abbreviation mechanism in Lex, whereby a symbol stands for some regular expression, is convenient enough that it is tempting to use it in interesting ways:

digits = [0 – 9]+
sum = (*digits* "+")* *digits*

These regular expressions define sums of the form `28+301+9`.

But now consider

digits = [0 – 9]+
sum = *expr* "+" *expr*
expr = "(" *sum* ")" | *digits*

This is meant to define expressions of the form:

```
(109+23)
61
(1+(250+3))
```

in which all the parentheses are balanced. But it is impossible for a finite automaton to recognize balanced parentheses (because a machine with N states cannot remember a parenthesis-nesting depth greater than N), so clearly *sum* and *expr* cannot be regular expressions.

So how does Lex manage to implement regular-expression abbreviations such as `digits`? The answer is that the right-hand-side `([0-9]+)` is simply

substituted for `digits` wherever it appears in regular expressions, *before* translation to a finite automaton.

This is not possible for the *sum*-and-*expr* language; we can first substitute *sum* into *expr*, yielding

expr = " (" *expr* "+" *expr* ") " | *digits*

but now an attempt to substitute expr into itself leads to

expr = " (" (" (" *expr* "+" *expr* ") " | *digits*) "+" *expr* ") " | *digits*

and the right-hand side now has just as many occurrences of *expr* as it did before – in fact, it has more!

Thus, the notion of abbreviation does not add expressive power to the language of regular expressions – there are no additional languages that can be defined – unless the abbreviations are recursive (or mutually recursive, as are *sum* and *expr*).

The additional expressive power gained by recursion is just what we need for parsing. Also, once we have abbreviations with recursion, we do not need alternation except at the top level of expressions, because the definition

$$expr = ab(c \mid d)e$$

can always be rewritten using an auxiliary definition as

$$aux = c \mid d$$
$$expr = a\ b\ aux\ e$$

In fact, instead of using the alternation mark at all, we can just write several allowable expansions for the same symbol:

$$aux = c$$
$$aux = d$$
$$expr = a\ b\ aux\ e$$

The Kleene closure is not necessary, since we can rewrite it so that

$$expr = (a\ b\ c)*$$

becomes

$$expr = (a\ b\ c)\ expr$$
$$expr = \epsilon$$

1 $S \rightarrow S\ ;\ S$	4 $E \rightarrow \text{id}$	8 $L \rightarrow E$
2 $S \rightarrow \text{id} := E$	5 $E \rightarrow \text{num}$	9 $L \rightarrow L\ ,\ E$
3 $S \rightarrow \text{print}\ (\ L\ )$	6 $E \rightarrow E + E$	
	7 $E \rightarrow (S\ ,\ E\ )$	

GRAMMAR 3.1. A syntax for straight-line programs.

What we have left is a very simple notation, called *context-free grammars*. Just as regular expressions can be used to define lexical structure in a static, declarative way, grammars define syntactic structure declaratively. But we will need something more powerful than finite automata to parse languages described by grammars.

In fact, grammars can also be used to describe the structure of lexical tokens, although regular expressions are adequate – and more concise – for that purpose.

3.1 CONTEXT-FREE GRAMMARS

As before, we say that a *language* is a set of *strings*; each string is a finite sequence of *symbols* taken from a finite *alphabet*. For parsing, the strings are source programs, the symbols are lexical tokens, and the alphabet is the set of token types returned by the lexical analyzer.

A context-free grammar describes a language. A grammar has a set of *productions* of the form

$$symbol \rightarrow symbol\ symbol \cdots symbol$$

where there are zero or more symbols on the right-hand side. Each symbol is either *terminal*, meaning that it is a token from the alphabet of strings in the language, or *nonterminal*, meaning that it appears on the left-hand side of some production. No token can ever appear on the left-hand side of a production. Finally, one of the nonterminals is distinguished as the *start symbol* of the grammar.

Grammar 3.1 is an example of a grammar for straight-line programs. The start symbol is S (when the start symbol is not written explicitly it is conventional to assume that the left-hand nonterminal in the first production is the start symbol). The terminal symbols are

id print num , + ( ) := ;

$\underline{S}$
S ; $\underline{S}$
$\underline{S}$; id : = E
id : = $\underline{E}$; id : = E
id : = num ; id : = $\underline{E}$
id : = num ; id : = E + $\underline{E}$
id : = num ; id : = $\underline{E}$ + (S , E)
id : = num ; id : = id + ($\underline{S}$, E)
id : = num ; id : = id + (id : = $\underline{E}$, E)
id : = num ; id : = id + (id : = E + E , $\underline{E}$)
id : = num ; id : = id + (id : = $\underline{E}$ + E , id)
id : = num ; id : = id + (id : = num + $\underline{E}$, id)
id : = num ; id : = id + (id : = num + num , id)

DERIVATION 3.2.

and the nonterminals are S, E, and L. One sentence in the language of this grammar is

```
id := num; id := id + (id := num + num, id)
```

where the source text (before lexical analysis) might have been

```
a := 7;
b := c + (d := 5 + 6, d)
```

The token-types (terminal symbols) are `id`, `num`, `:=`, and so on; the names (`a`, `b`, `c`, `d`) and numbers (7, 5, 6) are *semantic values* associated with some of the tokens.

DERIVATIONS

To show that this sentence is in the language of the grammar, we can perform a *derivation*: start with the start symbol, then repeatedly replace any nonterminal by one of its right-hand sides, as shown in Derivation 3.2.

There are many different derivations of the same sentence. A *leftmost* derivation is one in which the leftmost nonterminal symbol is always the one expanded; in a *rightmost* derivation, the rightmost nonterminal is always next to be expanded.

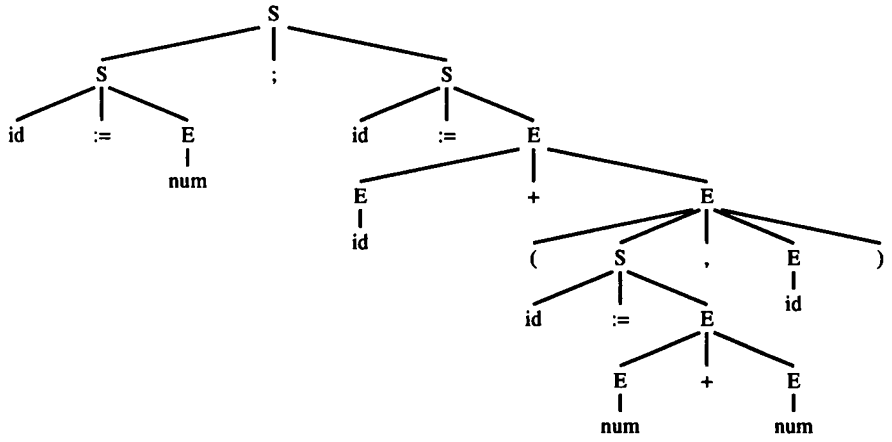


FIGURE 3.3. Parse tree.

Derivation 3.2 is neither leftmost nor rightmost; a leftmost derivation for this sentence would begin,

$$
\begin{aligned}
&\underline{S}\\
&\underline{S}\ ;\ S\\
&\text{id} := \underline{E}\ ;\ S\\
&\text{id} := \text{num}\ ;\ \underline{S}\\
&\text{id} := \text{num}\ ;\ \text{id} := \underline{E}\\
&\text{id} := \text{num}\ ;\ \text{id} := \underline{E} + E\\
&\quad\vdots
\end{aligned}
$$

PARSE TREES

A *parse tree* is made by connecting each symbol in a derivation to the one from which it was derived, as shown in Figure 3.3. Two different derivations can have the same parse tree.

AMBIGUOUS GRAMMARS

A grammar is *ambiguous* if it can derive a sentence with two different parse trees. Grammar 3.1 is ambiguous, since the sentence `id := id+id+id` has two parse trees (Figure 3.4).

Grammar 3.5 is also ambiguous; Figure 3.6 shows two parse trees for the sentence `1-2-3`, and Figure 3.7 shows two trees for `1+2*3`. Clearly, if we use

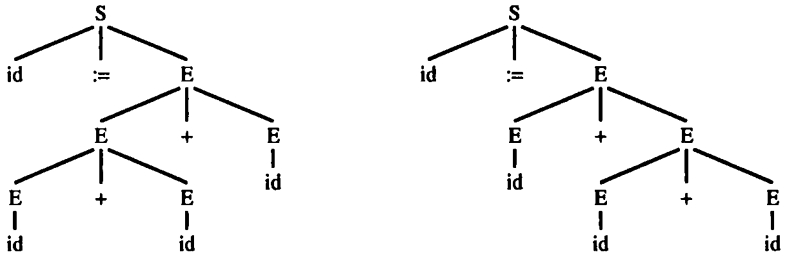


FIGURE 3.4. Two parse trees for the same sentence using Grammar 3.1.

$E \rightarrow$ id
$E \rightarrow$ num
$E \rightarrow E * E$
$E \rightarrow E / E$
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow (E)$

GRAMMAR 3.5.

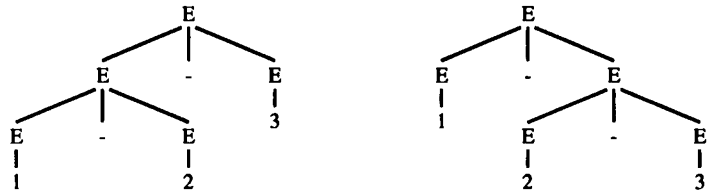


FIGURE 3.6. Two parse trees for the sentence 1-2-3 in Grammar 3.5.

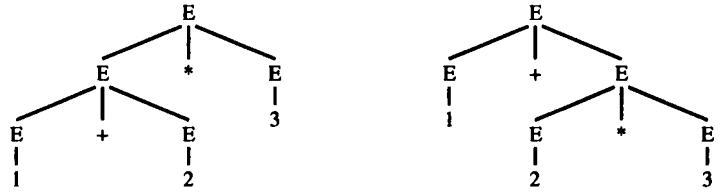


FIGURE 3.7. Two parse trees for the sentence 1+2*3 in Grammar 3.5.

$E \rightarrow E + T$	$T \rightarrow T * F$	$F \rightarrow$ id
$E \rightarrow E - T$	$T \rightarrow T / F$	$F \rightarrow$ num
$E \rightarrow T$	$T \rightarrow F$	$F \rightarrow (E)$

GRAMMAR 3.8.

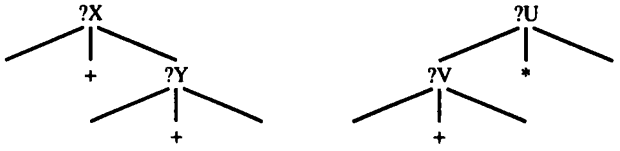


FIGURE 3.9. Parse trees that Grammar 3.8 will never produce.

parse trees to interpret the meaning of the expressions, the two parse trees for `1-2-3` mean different things: $(1 - 2) - 3 = -4$ versus $1 - (2 - 3) = 2$. Similarly, $(1 + 2) \times 3$ is not the same as $1 + (2 \times 3)$. And indeed, compilers do use parse trees to derive meaning.

Therefore, ambiguous grammars are problematic for compiling: in general we would prefer to have unambiguous grammars. Fortunately, we can often transform ambiguous grammars to unambiguous grammars.

Let us find an unambigous grammar that accepts the same language as Grammar 3.5. First, we would like to say that * *binds tighter* than +, or has *higher precedence*. Second, we want to say that each operator *associates to the left*, so that we get $(1 - 2) - 3$ instead of $1 - (2 - 3)$. We do this by introducing new nonterminal symbols to get Grammar 3.8.

The symbols E, T, and F stand for *expression*, *term*, and *factor*; conventionally, factors are things you multiply and terms are things you add.

This grammar accepts the same set of sentences as the ambiguous grammar, but now each sentence has exactly one parse tree. Grammar 3.8 can never produce parse trees of the form shown in Figure 3.9 (see Exercise 3.17).

Had we wanted to make * associate to the right, we could have written its production as $T \rightarrow F * T$.

We can usually eliminate ambiguity by transforming the grammar. Though there are some languages (sets of strings) that have ambiguous grammars but no unambiguous grammar, such languages may be problematic as *programming* languages because the syntactic ambiguity may lead to problems in writing and understanding programs.

$S \rightarrow E\ \$$

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$

$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$

$F \rightarrow \text{id}$
$F \rightarrow \text{num}$
$F \rightarrow (E)$

GRAMMAR 3.10.

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$
$S \rightarrow \text{begin } S\ L$
$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$
$L \rightarrow ;\ S\ L$

$E \rightarrow \text{num} = \text{num}$

GRAMMAR 3.11.

END-OF-FILE MARKER

Parsers must read not only terminal symbols such as +, -, num, and so on, but also the end-of-file marker. We will use $ to represent end of file.

Suppose S is the start symbol of a grammar. To indicate that $ must come after a complete S-phrase, we augment the grammar with a new start symbol S' and a new production $S' \rightarrow S\$$.

In Grammar 3.8, E is the start symbol, so an augmented grammar is Grammar 3.10.

3.2 PREDICTIVE PARSING

Some grammars are easy to parse using a simple algorithm known as *recursive descent*. In essence, each grammar production turns into one clause of a recursive function. We illustrate this by writing a recursive-descent parser for Grammar 3.11.

A recursive-descent parser for this language has one function for each nonterminal and one clause for each production.

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();
                                  eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();
       }}
void L(void) {switch(tok) {
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break;
        default:    error();
       }}
void E(void) {  eat(NUM); eat(EQ); eat(NUM); }
```

With suitable definitions of `error` and `getToken`, this program will parse very nicely.

Emboldened by success with this simple method, let us try it with Grammar 3.10:

```
void S(void) {  E(); eat(EOF); }
void E(void) {switch (tok) {
          case ?: E(); eat(PLUS); T(); break;
          case ?: E(); eat(MINUS); T(); break;
          case ?: T(); break;
          default: error();
         }}
void T(void) {switch (tok) {
          case ?: T(); eat(TIMES); F(); break;
          case ?: T(); eat(DIV); F(); break;
          case ?: F(); break;
          default: error();
         }}
```

There is a *conflict* here: the E function has no way to know which clause to use. Consider the strings `(1*2-3)+4` and `(1*2-3)`. In the former case, the initial call to E should use the $E \rightarrow E + T$ production, but the latter case should use $E \rightarrow T$.

Recursive-descent, or *predictive*, parsing works only on grammars where the *first terminal symbol* of each subexpression provides enough information

$Z \rightarrow d$	$Y \rightarrow$	$X \rightarrow Y$
$Z \rightarrow X\ Y\ Z$	$Y \rightarrow c$	$X \rightarrow a$

GRAMMAR 3.12.

to choose which production to use. To understand this better, we will formalize the notion of FIRST sets, and then derive conflict-free recursive-descent parsers using a simple algorithm.

Just as lexical analyzers can be constructed from regular expressions, there are parser-generator tools that build predictive parsers. But if we are going to use a tool, then we might as well use one based on the more powerful LR(1) parsing algorithm, which will be described in Section 3.3.

Sometimes it's inconvenient or impossible to use a parser-generator tool. The advantage of predictive parsing is that the algorithm is simple enough that we can use it to construct parsers by hand – we don't need automatic tools.

FIRST AND FOLLOW SETS

Given a string γ of terminal and nonterminal symbols, FIRST(γ) is the set of all terminal symbols that can begin any string derived from γ. For example, let $\gamma = T * F$. Any string of terminal symbols derived from γ must start with `id`, `num`, or `(`. Thus,

FIRST($T * F$) = {`id`, `num`, `(`}.

If two different productions $X \rightarrow \gamma_1$ and $X \rightarrow \gamma_2$ have the same left-hand-side symbol (X) and their right-hand sides have overlapping FIRST sets, then the grammar cannot be parsed using predictive parsing. If some terminal symbol I is in FIRST(γ_1) and also in FIRST(γ_2), then the X function in a recursive-descent parser will not know what to do if the input token is I.

The computation of FIRST sets looks very simple: if $\gamma = X\,Y\,Z$, it seems as if Y and Z can be ignored, and FIRST(X) is the only thing that matters. But consider Grammar 3.12. Because Y can produce the empty string – and therefore X can produce the empty string – we find that FIRST($X\,Y\,Z$) must include FIRST(Z). Therefore, in computing FIRST sets, we must keep track of which symbols can produce the empty string; we say such symbols are *nullable*. And we must keep track of what might follow a nullable symbol.

With respect to a particular grammar, given a string γ of terminals and nonterminals,

- nullable(X) is true if X can derive the empty string.
- FIRST(γ) is the set of terminals that can begin strings derived from γ.
- FOLLOW(X) is the set of terminals that can immediately follow X. That is, $t \in$ FOLLOW(X) if there is any derivation containing Xt. This can occur if the derivation contains $X\,Y\,Z\,t$ where Y and Z both derive ϵ.

A precise definition of FIRST, FOLLOW, and nullable is that they are the smallest sets for which these properties hold:

For each terminal symbol Z, FIRST[Z] = {Z}.
for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
  for each i from 1 to k, each j from $i+1$ to k,
    if all the Y_i are nullable
      then nullable[X] = true
    if $Y_1 \cdots Y_{i-1}$ are all nullable
      then FIRST[X] = FIRST[X] $\cup$ FIRST[Y_i]
    if $Y_{i+1} \cdots Y_k$ are all nullable
      then FOLLOW[Y_i] = FOLLOW[Y_i] $\cup$ FOLLOW[X]
    if $Y_{i+1} \cdots Y_{j-1}$ are all nullable
      then FOLLOW[Y_i] = FOLLOW[Y_i] $\cup$ FIRST[Y_j]

Algorithm 3.13 for computing FIRST, FOLLOW, and nullable just follows from these facts; we simply replace each equation with an assignment statement, and iterate.

Of course, to make this algorithm efficient it helps to examine the productions in the right order; see Section 17.4. Also, the three relations need not be computed simultaneously; nullable can be computed by itself, then FIRST, then FOLLOW.

This is not the first time that a group of equations on sets has become the algorithm for calculating those sets; recall the algorithm on page 28 for computing ϵ-closure. Nor will it be the last time; the technique of iteration to a fixed point is applicable in dataflow analysis for optimization, in the back end of a compiler.

We can apply this algorithm to Grammar 3.12. Initially, we have:

	nullable	FIRST	FOLLOW
X	no		
Y	no		
Z	no		

In the first iteration, we find that $a \in$ FIRST[X], Y is nullable, $c \in$ FIRST[Y], $d \in$ FIRST[Z], $d \in$ FOLLOW[X], $c \in$ FOLLOW[X],

Algorithm to compute FIRST, FOLLOW, *and* nullable.
Initialize FIRST and FOLLOW to all empty sets, and nullable to all false.
for each terminal symbol Z
  FIRST[Z] $\leftarrow$ {Z}
repeat
  for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
    for each i from 1 to k, each j from $i+1$ to k,
      if all the Y_i are nullable
        then nullable[X] $\leftarrow$ true
      if $Y_1 \cdots Y_{i-1}$ are all nullable
        then FIRST[X] $\leftarrow$ FIRST[X] $\cup$ FIRST[Y_i]
      if $Y_{i+1} \cdots Y_k$ are all nullable
        then FOLLOW[Y_i] $\leftarrow$ FOLLOW[Y_i] $\cup$ FOLLOW[X]
      if $Y_{i+1} \cdots Y_{j-1}$ are all nullable
        then FOLLOW[Y_i] $\leftarrow$ FOLLOW[Y_i] $\cup$ FIRST[Y_j]
until FIRST, FOLLOW, and nullable did not change in this iteration.

ALGORITHM 3.13. Iterative computation of *FIRST, FOLLOW,* and *nullable.*

$d \in$ FOLLOW[Y]. Thus:

	nullable	FIRST	FOLLOW
X	no	a	c d
Y	yes	c	d
Z	no	d	

In the second iteration, we find that X is nullable, $c \in$ FIRST[X], $\{a, c\} \subseteq$ FIRST[Z], $\{a, c, d\} \subseteq$ FOLLOW[X], $\{a, c, d\} \subseteq$ FOLLOW[Y]. Thus:

	nullable	FIRST	FOLLOW
X	yes	a c	a c d
Y	yes	c	a c d
Z	no	a c d	

The third iteration finds no new information, and the algorithm terminates.

It is useful to generalize the FIRST relation to strings of symbols:

FIRST($X\gamma$) = FIRST[X] if not nullable[X]
FIRST($X\gamma$) = FIRST[X] $\cup$ FIRST(γ) if nullable[X]

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$

FIGURE 3.14. Predictive parsing table for Grammar 3.12.

and similarly, we say that a string γ is nullable if each symbol in γ is nullable.

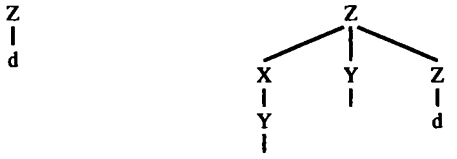
CONSTRUCTING A PREDICTIVE PARSER

Consider a recursive-descent parser. The parsing function for some nonterminal X has a clause for each X-production; it must choose one of these clauses based on the next token T of the input. If we can choose the right production for each (X, T), then we can write the recursive-descent parser. All the information we need can be encoded as a two-dimensional table of productions, indexed by nonterminals X and terminals T. This is called a *predictive parsing* table.

To construct this table, enter production $X \to \gamma$ in row X, column T of the table for each $T \in \text{FIRST}(\gamma)$. Also, if γ is nullable, enter the production in row X, column T for each $T \in \text{FOLLOW}[X]$.

Figure 3.14 shows the predictive parser for Grammar 3.12. But some of the entries contain more than one production! The presence of duplicate entries means that predictive parsing will not work on Grammar 3.12.

If we examine the grammar more closely, we find that it is ambiguous. The sentence d has many parse trees, including:



An ambiguous grammar will always lead to duplicate entries in a predictive parsing table. If we need to use the language of Grammar 3.12 as a programming language, we will need to find an unambiguous grammar.

Grammars whose predictive parsing tables contain no duplicate entries

are called LL(1). This stands for *Left-to-right parse, Leftmost-derivation, 1-symbol lookahead.* Clearly a recursive-descent (predictive) parser examines the input left-to-right in one pass (some parsing algorithms do not, but these are generally not useful for compilers). The order in which a predictive parser expands nonterminals into right-hand sides (that is, the recursive-descent parser calls functions corresponding to nonterminals) is just the order in which a leftmost derivation expands nonterminals. And a recursive-descent parser does its job just by looking at the next token of the input, never looking more than one token ahead.

We can generalize the notion of FIRST sets to describe the first k tokens of a string, and to make an LL(k) parsing table whose rows are the nonterminals and columns are every sequence of k terminals. This is rarely done (because the tables are so large), but sometimes when you write a recursive-descent parser by hand you a need to look more than one token ahead.

Grammars parsable with LL(2) parsing tables are called LL(2) grammars, and similarly for LL(3), etc. Every LL(1) grammar is an LL(2) grammar, and so on. No ambiguous grammar is LL(k) for any k.

ELIMINATING LEFT RECURSION

Suppose we want to build a predictive parser for Grammar 3.10. The two productions

$$E \rightarrow E + T$$
$$E \rightarrow T$$

are certain to cause duplicate entries in the LL(1) parsing table, since any token in FIRST(T) will also be in FIRST($E + T$). The problem is that E appears as the first right-hand-side symbol in an E-production; this is called *left recursion.* Grammars with left recursion cannot be LL(1).

To eliminate left recursion, we will rewrite using right recursion. We introduce a new nonterminal E', and write

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow$$

This derives the same set of strings (on T and +) as the original two productions, but now there is no left recursion.

$S \rightarrow E\ \$$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$
$E' \rightarrow -\ T\ E'$
$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$
$T' \rightarrow /\ F\ T'$
$T' \rightarrow$

$F \rightarrow \text{id}$
$F \rightarrow \text{num}$
$F \rightarrow (\ E\ )$

GRAMMAR 3.15.

	nullable	FIRST	FOLLOW
S	no	(id num	
E	no	(id num	) $
E'	yes	+ -	) $
T	no	(id num	) + - $
T'	yes	* /	) + - $
F	no	(id num	) * / + - $

TABLE 3.16. Nullable, FIRST, and FOLLOW for Grammar 3.8.

In general, whenever we have productions $X \rightarrow X\gamma$ and $X \rightarrow \alpha$, where α does not start with X, we know that this derives strings of the form $\alpha\gamma*$, an α followed by zero or more γ. So we can rewrite the regular expression using right recursion:

$$\begin{pmatrix} X \rightarrow X\ \gamma_1 \\ X \rightarrow X\ \gamma_2 \\ X \rightarrow \alpha_1 \\ X \rightarrow \alpha_2 \end{pmatrix} \Longrightarrow \begin{pmatrix} X \rightarrow \alpha_1\ X' \\ X \rightarrow \alpha_2\ X' \\ X' \rightarrow \gamma_1\ X' \\ X' \rightarrow \gamma_2\ X' \\ X' \rightarrow \end{pmatrix}$$

Applying this transformation to Grammar 3.10, we obtain Grammar 3.15.

To build a predictive parser, first we compute nullable, FIRST, and FOLLOW (Table 3.16). The predictive parser for Grammar 3.15 is shown in Table 3.17.

LEFT FACTORING

We have seen that left recursion interferes with predictive parsing, and that it can be eliminated. A similar problem occurs when two productions for the

	+	*	id	(	)	$
S			$S \rightarrow E\$$	$S \rightarrow E\$$		
E			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'	$E' \rightarrow +TE'$				$E' \rightarrow$	$E' \rightarrow$
T			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'	$T' \rightarrow$	$T' \rightarrow *FT'$			$T' \rightarrow$	$T' \rightarrow$
F			$F \rightarrow$ id	$F \rightarrow (E)$		

TABLE 3.17. Predictive parsing table for Grammar 3.15. We omit the columns for num, /, and -, as they are similar to others in the table.

same nonterminal start with the same symbols. For example:

$S \rightarrow$ if E then S else S
$S \rightarrow$ if E then S

In such a case, we can *left factor* the grammar – that is, take the allowable endings ("else S" and ϵ) and make a new nonterminal X to stand for them:

$S \rightarrow$ if E then S X
$X \rightarrow$
$X \rightarrow$ else S

The resulting productions will not pose a problem for a predictive parser.

ERROR RECOVERY

Armed with a predictive parsing table, it is easy to write a recursive-descent parser. Here is a representative fragment of a parser for Grammar 3.15:

```
void T(void) {switch (tok) {
        case ID:
        case NUM:
        case LPAREN:  F(); Tprime(); break;
        default:  error!
      }}

void Tprime(void) {switch (tok) {
        case PLUS:    break;
        case TIMES:   eat(TIMES); F(); Tprime(); break;
        case EOF:     break;
        case RPAREN:  break;
        default:  error!
      }}
```

A blank entry in row T, column x of the LL(1) parsing table indicates that the parsing function `T()` does not expect to see token x – this will be a syntax error. How should *error* be handled? It is safe just to raise an exception and quit parsing, but this is not very friendly to the user. It is better to print an error message and recover from the error, so that other syntax errors can be found in the same compilation.

A syntax error occurs when the string of input tokens is not a sentence in the language. Error recovery is a way of finding some sentence similar to that string of tokens. This can proceed by deleting, replacing, or inserting tokens.

For example, error recovery for `T` could proceed by inserting a `num` token. It's not necessary to adjust the actual input; it suffices to pretend that the `num` was there, print a message, and return normally.

```
void T(void) {switch (tok) {
        case ID:
        case NUM:
        case LPAREN: F(); Tprime(); break;
        default:  printf("expected id, num, or left-paren");
      }}
```

It's a bit dangerous to do error recovery by insertion, because if the error cascades to produce another error, the process might loop infinitely. Error recovery by deletion is safer, because the loop must eventually terminate when end-of-file is reached.

Simple recovery by deletion works by skipping tokens until a token in the FOLLOW set is reached. For example, error recovery for T' could work like this:

```
int Tprime_follow [] = {PLUS, TIMES, RPAREN, EOF, -1};

void Tprime(void) { switch (tok) {
        case PLUS:   break;
        case TIMES:  eat(TIMES); F(); Tprime(); break;
        case RPAREN: break;
        case EOF:    break;
        default:   printf("expected +, *, right-paren,
                           or end-of-file");
                   skipto(Tprime_follow);
      }}
```

A recursive-descent parser's error-recovery mechanisms must be adjusted (sometimes by trial and error) to avoid a long cascade of error-repair messages resulting from a single token out of place.

3.3 LR PARSING

The weakness of LL(k) parsing techniques is that they must *predict* which production to use, having seen only the first k tokens of the right-hand side. A more powerful technique, LR(k) parsing, is able to postpone the decision until it has seen input tokens corresponding to the entire right-hand side of the production in question (and k more input tokens beyond).

LR(k) stands for *Left-to-right parse, Rightmost-derivation, k-token lookahead.* The use of a rightmost derivation seems odd; how is that compatible with a left-to-right parse? Figure 3.18 illustrates an LR parse of the program

```
a := 7;
b := c + (d := 5 + 6, d)
```

using Grammar 3.1, augmented with a new start production $S' \rightarrow S\$$.

The parser has a *stack* and an *input*. The first k tokens of the input are the *lookahead*. Based on the contents of the stack and the lookahead, the parser performs two kinds of actions:

Shift: move the first input token to the top of the stack.
Reduce: Choose a grammar rule $X \rightarrow A\ B\ C$; pop C, B, A from the top of the stack; push X onto the stack.

Initially, the stack is empty and the parser is at the beginning of the input. The action of shifting the end-of-file marker \$ is called *accepting* and causes the parser to stop successfully.

In Figure 3.18, the stack and input are shown after every step, along with an indication of which action has just been performed. The concatenation of stack and input is always one line of a rightmost derivation; in fact, Figure 3.18 shows the rightmost derivation of the input string, upside-down.

LR PARSING ENGINE

How does the LR parser know when to shift and when to reduce? By using a deterministic finite automaton! The DFA is not applied to the input – finite automata are too weak to parse context-free grammars – but to the stack. The edges of the DFA are labeled by the symbols (terminals and nonterminals) that can appear on the stack. Table 3.19 is the transition table for Grammar 3.1.

The elements in the transition table are labeled with four kinds of actions: