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 ML-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 ML-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

$$expr = \text{“(”}\ expr\ \text{“+”}\ expr\ \text{“)”} \mid digits$$

but now an attempt to substitute expr into itself leads to

$$expr = \text{“(”}\ (\ \text{“(”}\ expr\ \text{“+”}\ expr\ \text{“)”} \mid digits\ )\ \text{“+”}\ expr\ \text{“)”} \mid digits$$

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

$$\begin{aligned} aux &= c \mid d \\ expr &= a\ b\ aux\ e \end{aligned}$$

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

$$\begin{aligned} aux &= c \\ aux &= d \\ expr &= a\ b\ aux\ e \end{aligned}$$

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

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

becomes

$$\begin{aligned} expr &= (a\ b\ c)\ expr \\ expr &= \epsilon \end{aligned}$$

1 $S \rightarrow S\ ;\ S$	4 $E \rightarrow$ id	8 $L \rightarrow E$
2 $S \rightarrow$ id $:=$ E	5 $E \rightarrow$ num	9 $L \rightarrow L\ ,\ E$
3 $S \rightarrow$ 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.

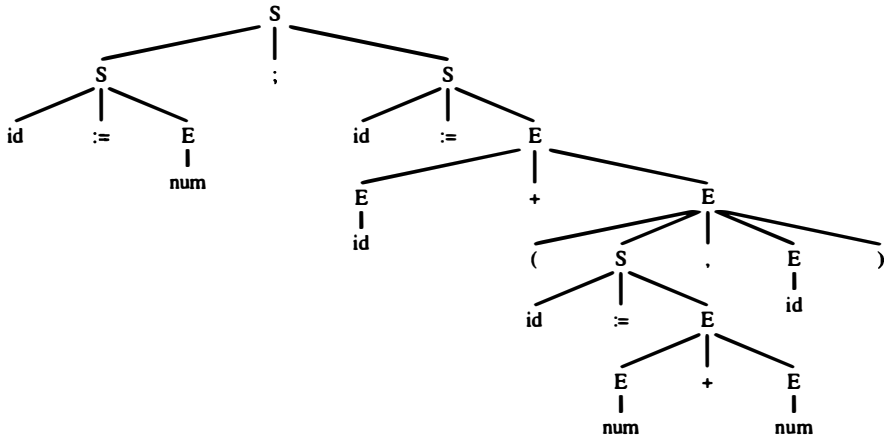

FIGURE 3.3. Parse tree.

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

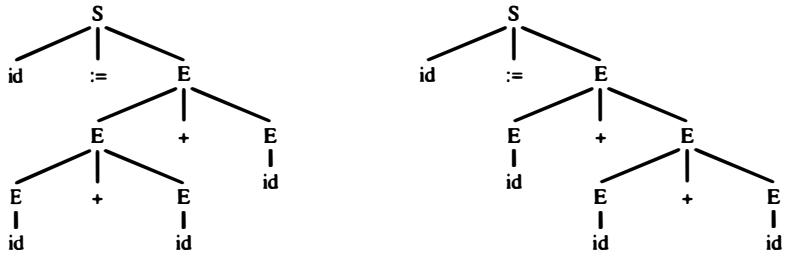

FIGURE 3.4. Two parse trees for the same sentence using Grammar 3.1.

$E \rightarrow$ 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.

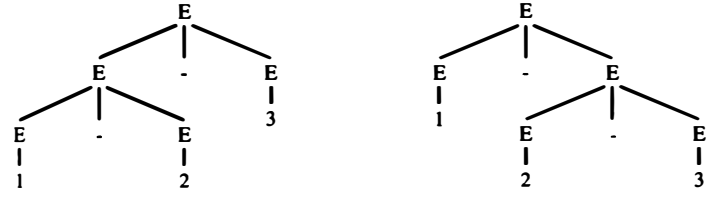

FIGURE 3.6. Two parse trees for the sentence `1-2-3` in Grammar 3.5.

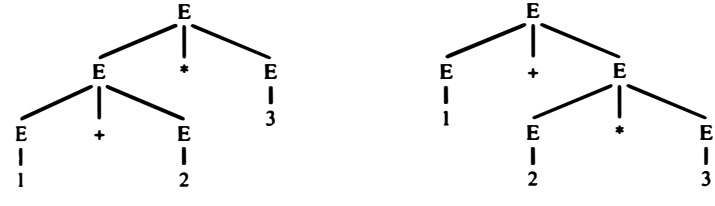

FIGURE 3.7. Two parse trees for the sentence `1+2*3` in Grammar 3.5.

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

GRAMMAR 3.8.

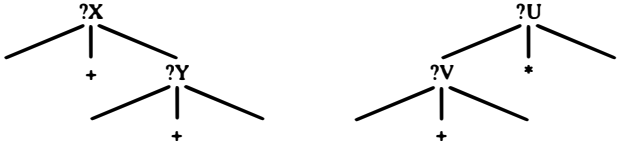

FIGURE 3.9. Parse trees that Grammar 3.8 will never produce.

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.