Context-Free Languages

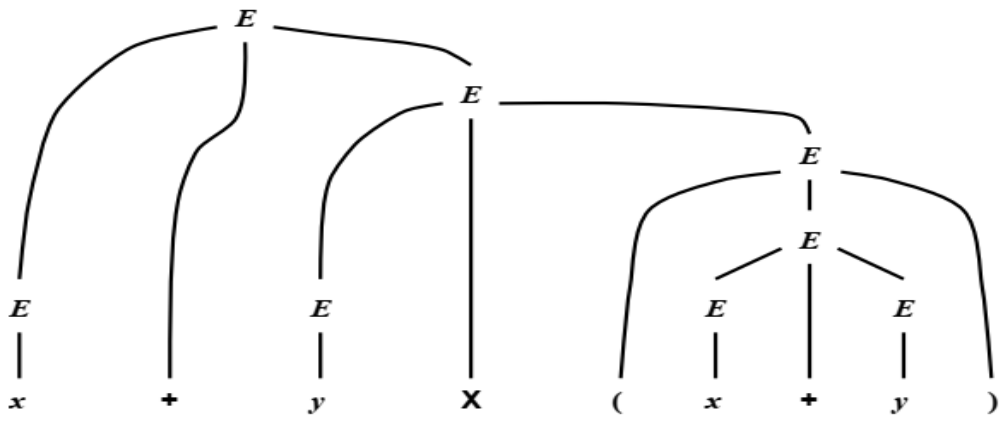
Context-Free Grammars

As we saw at the end of the previous chapter, one feature of programming languages that makes them nonregular is the arbitrary nesting of parenthesis in expressions. Regular expressions cannot be used to define what syntactically correct expressions look like. "Context-free grammars" can.

An example of a context-free grammar is this:

$$
\begin{aligned}
E &\rightarrow E + E \\
E &\rightarrow E \times E \\
E &\rightarrow (E) \\
E &\rightarrow x \\
E &\rightarrow y
\end{aligned}
$$

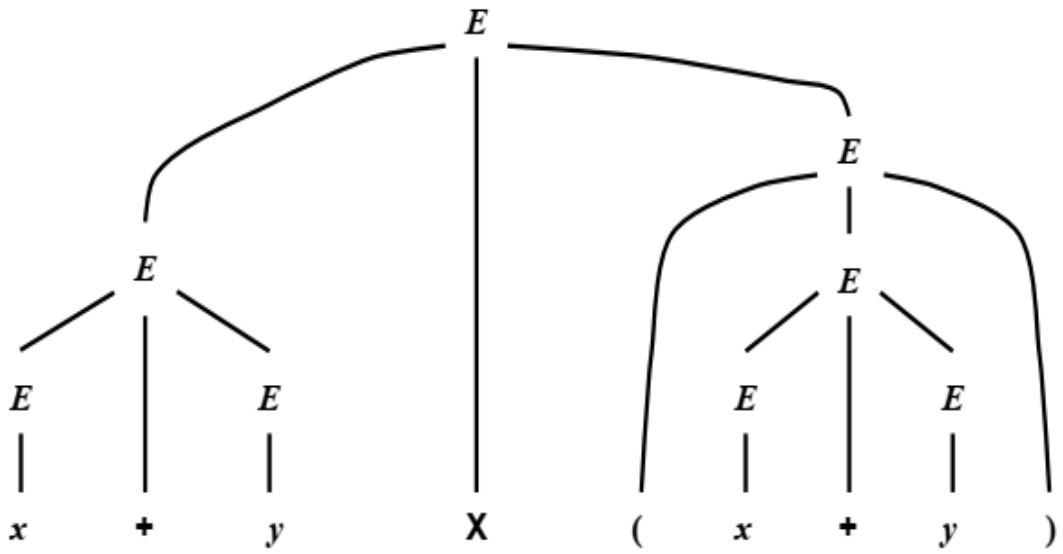
The way this grammar — call it G_1 — is used is to start with the symbol on left of the arrow in the first line, E. This is the "start symbol" — the string of length 1 we start out with. We then grow that string by repeatedly replacing what's on the left-hand side of an arrow with what's on the right-hand side. The best way to keep track is a tree:



There are choices of which replacements to make. The choices made above resulted in a tree whose leaves, read from left to right, read $x+y\times(x+y)$. The grammar G_1 is said to "derive" that string: $x+y\times(x+y)\in L(G_1)$. The tree is a "derivation tree" for the string.

Derivation trees are also called "parse trees." They are used in compilers to know how to translate a given expression, or a whole program, into machine language, and in interpreters to know how to evaluate an expression, or interpret a whole program.

Our sample grammar does not serve that purpose well, though. This is because it allows a second derivation tree for the same string:

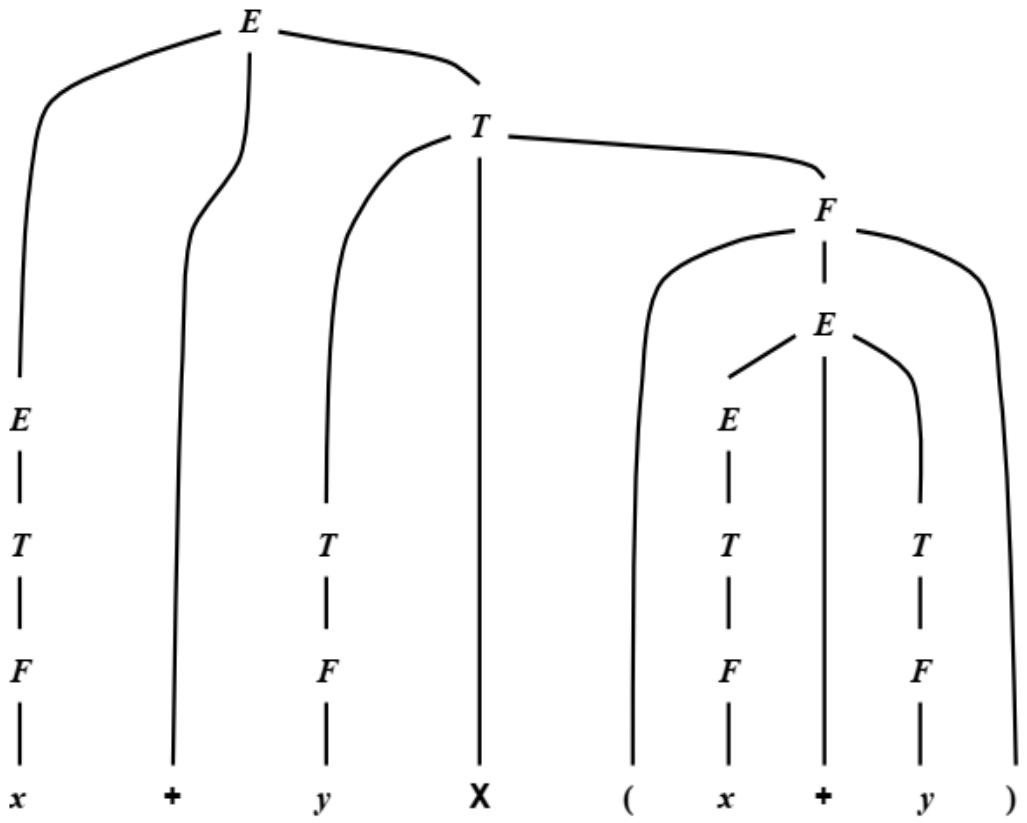


Such a grammar is said to be "ambiguous." The two parse trees for the string $x+y\times(x+y)$ lead to two different evaluations of the expression. If we plug in $x=1$ and $y=5$, the first tree leads to a value of 36 for the expression, whereas the second one leads to a value of 31. Of the two trees, only the first one matches our conventions about the precedence of arithmetic operations. The first one is "right," the second one is "wrong." The problem is, this grammar G_1, while it does perfectly well describe the language in question, does not help us define the one and only "correct" way to parse each string in the language. The following

grammar G_2 does do that:

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow T \times F \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow x \\ F &\rightarrow y \end{aligned}$$

This grammar only allows one derivation tree, the "right" one, for any arithmetic expressions involving $+$, $\times$, parentheses, and the variables x and y. The tree for the expression $x + y \times (x + y)$ is:



Parsing

Recursive-Descent Parsing

A grammar like G_2 above is easily turned into a parser, i.e., a program that determines whether or not a given input string is in the language and does so (necessarily) by building a parse tree, implicitly or explicitly. See Figure 3.1. The program is nondeterministic, though — certainly a flaw we do not want to tolerate for very long.

In what sense does this program "parse" expressions?

```
main ( )
{
    E();
    if (getchar() == EOF)
        ACCEPT;
    else
        REJECT;
}

E ( )
{
    nondet-branch:
        {E(); terminal('+'); T();}
        {T();}
}

T ( )
{
    nondet-branch:
        {T(); terminal('*'); F();}
        {F();}
}

F ( )
{
    nondet-branch:
        {terminal('('); E(); terminal(')');}
        {terminal('x');}
        {terminal('y');}
}

terminal( char c )
{
    if (getchar() != c)
        REJECT;
}
```

Figure 3.1: A nondeterministic recursive-descent parser

Being a nondeterministic program, the program as a whole is said to accept an input string if there is at least one path of execution that accepts it. Consider again the string $x+y\times(x+y)$. The one path of execution that results in acceptance is the one where the first execution of `E` in turn calls `E`, `terminal('+')`, and `T`. These calls correspond to the children of the root of the derivation tree. If we chart the calling pattern of the accepting execution we end up drawing the parse tree. *The accepting computation looks like a preorder traversal of the parse tree of the input string.*

The above construction shows how to turn any context-free grammar into a program that is like the finite programs of Chapter 2 except that it uses two additional features: recursion and nondeterminism. Such programs are often called "pushdown machines" or "pushdown automata (pda's)." They are equivalent to finite-state machines augmented by a stack to implement recursion. Thus the above example generalizes to the following theorem.

Theorem 17 *For every context-free grammar G there is a pushdown machine M with $L(M) = L(G)$.*

The converse is also true, providing a complete characterization of context-free languages in terms of a type of program.

Theorem 18 *For every pushdown machine M there is a context-free grammar G with $L(M) = L(G)$.*

(We don't prove this here.)

Deterministic Parsing

Intrinsically Nondeterministic Context-Free Languages

With regular languages and finite-state machines it was always possible to remove the nondeterminism from the machines. The trick was to replace states by sets of states. This worked because the power set (set of subsets) of a finite set was again a finite set. This approach does not work for pushdown machines because a set of stacks, or even just two stacks, cannot be implemented with a single stack. Without giving any proof of this, here is an example of a grammar that derives an intrinsically nondeterministic language. The language is very simple, all bitstrings of odd length whose middle character is a 1.

$$\begin{aligned} S &\rightarrow B\ S\ B \mid 1 \\ B &\rightarrow 0 \mid 1 \end{aligned}$$

To parse this language with a pushdown machine, nondeterminism is needed. We do not prove this here, but the intuition is that the parser must keep building up a stack of recursive calls for the first half of the input string and climb out of the recursion during the second half. The problem is, with information limited by a constant amount, it is not possible to know when the second half starts.

A "Predictive" Recursive-Descent Parser

(The example is after Aho and Ullman, *Principles of Compiler Design*, 1978, p.176ff.)
While a deterministic recursive-descent parser does not exist for all context-free languages, some languages are simple enough to yield to such an approach. For example, the grammar for arithmetic expressions used earlier can be transformed, without changing the language, into the following:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \lambda \\ T &\rightarrow FT' \\ T' &\rightarrow \times FT' \mid \lambda \\ F &\rightarrow (E) \\ F &\rightarrow x \\ F &\rightarrow y \end{aligned}$$

The work done by the first two productions version, which was to generate intermediate strings of the form

$$T + T + \cdots + T$$

is done by the first two productions of the new version. The new grammar leads to a parser in which the decision which of the nondeterministic paths has a chance of leading to acceptance of the input can be made on the basis of looking at the next character of the input. This is so because whenever the grammar has several right-hand sides for the same left-hand symbol, the right-hand sides differ in their first character. The resulting deterministic parser in shown in Figure 3.2.

A General Parsing Algorithm

Instead of using nondeterminism, which is impractical even when implemented by a deterministic tree-traversal, one can use a "dynamic programming" approach to parse any context-free language. Dynamic programming is a type of algorithm somewhat resembling "divide-and-conquer," which inductively (or recursively, depending on the implementation) builds up information about subproblems. It typically has $O(n^3)$ time complexity.

There are two steps to using this technique. First, we change the grammar into "Chomsky Normal Form," then we apply the actual dynamic programming algorithm.

Chomsky Normal Form Every context-free grammar can be converted into an equivalent one in which each production is of one of the two forms

```
main ( )
{
    E();
    if (getchar() == EOF)
        ACCEPT;
    else
        REJECT;
}

E ( ) { T(); E'(); }

E' ( )
{
    if (NEXTCHAR == '+')
        {terminal('+'); T(); E'();}
    else
        // do nothing, just return
}

T ( ) { F(); T'(); }

T' ( )
{
    if (NEXTCHAR == '*')
        {terminal('*'); F(); T'();}
    else
        // do nothing, just return
}

F ( )
{
    switch (NEXTCHAR)
    {
        case '(': {terminal('('); E(); terminal(')');}
                  break;
        case 'x': terminal('x');
                  break;
        case 'y': terminal('y');
                  break;
        default:  REJECT;
    }
}

...
```

Figure 3.2: A deterministic recursive-descent parser

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

where A, B, C are syntactic variables, not necessarily different ones, and a is a terminal symbol. This form of a context-free grammar is called "Chomsky Normal Form." To see how this conversion can be done consider this production:

$$A \rightarrow BCdEfG \tag{3.1}$$

The production 3.1 can equivalently be replaced by the set of productions

$$\begin{aligned} A &\rightarrow BCDEFG \\ D &\rightarrow d \\ F &\rightarrow f \end{aligned} \tag{3.2}$$

and the production 3.2 can then be replaced by the set of productions

$$\begin{aligned} A &\rightarrow BC' \\ C' &\rightarrow CD' \\ D' &\rightarrow DE' \\ E' &\rightarrow EF' \\ F' &\rightarrow FG \end{aligned}$$

(The example below is from Hopcroft and Ullman, Introduction to Automata Theory, Languages, and Computation, 1979, pp. 140-1.)

CYK Parsing: An Example of "Dynamic Programming"

Figure 3.3 shows an example. The "*length = 2, starting position = 3*" field contains all those variables of the grammar from which one can derive the substring of length 2 that starts at position 3. (The string is ab. The variables are S and C.) The rows of the table get filled in from top (length=1) to bottom (length=5). The whole string is in the language if the start symbol of the grammar shows up in the bottom field. Filling in a field makes use of the contents of fields filled in earlier, in a pattern indicated by the two arrows in the Figure.

The algorithm is easily modified to yield a parse tree, or even sets of parse trees.