

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)

Next: [3.2 Predictive Parsing](#) Up: [3 Parsing](#) Previous: [3 Parsing](#) [Contents](#)

3.1 Context-free Grammars

Consider the following input string:

`x+2*y`

When scanned by a scanner, it produces the following stream of tokens:

```
id(x)
+
num(2)
*
id(y)
```

The goal is to parse this expression and construct a data structure (a parse tree) to represent it. One possible syntax for expressions is given by the following grammar G1:

```
E ::= E + T
    | E - T
    | T
T ::= T * F
    | T / F
    | F
F ::= num
    | id
```

where E , T and F stand for expression, term, and factor respectively. For example, the rule for E indicates that an expression E can take one of the following 3 forms: an expression followed by the token $+$ followed by a term, or an expression followed by the token $-$ followed by a term, or simply a term. The first rule for E is actually a shorthand of 3 productions:

```
E ::= E + T
E ::= E - T
E ::= T
```

G1 is an example of a context-free grammar (defined below); the symbols E , T and F are *nonterminals* and should be defined using production rules, while $+$, $-$, $*$, $/$, `num`, and `id` are *terminals* (ie, tokens) produced by the scanner. The nonterminal E is the *start symbol* of the grammar.

In general, a *context-free grammar* (CFG) has a finite set of terminals (tokens), a finite set of nonterminals from which one is the start symbol, and a finite set of *productions* of the form:

```
A ::= X1 X2 ... Xn
```

where A is a nonterminal and each x_i is either a terminal or nonterminal symbol.

Given two sequences of symbols a and b (can be any combination of terminals and nonterminals) and a

production $A ::= X_1X_2...X_n$, the form $aAb \Rightarrow aX_1X_2...X_nb$ is called a *derivation*. That is, the nonterminal symbol A is replaced by the rhs (right-hand-side) of the production for A . For example,

$T / F + 1 - x \Rightarrow T * F / F + 1 - x$

is a derivation since we used the production $T := T * F$.

Top-down parsing starts from the start symbol of the grammar S and applies derivations until the entire input string is derived (ie, a sequence of terminals that matches the input tokens). For example,

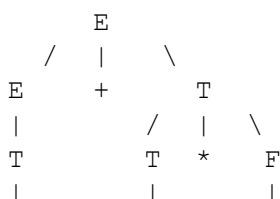
$E \Rightarrow E + T$
 $\Rightarrow E + T * F$
 $\Rightarrow T + T * F$
 $\Rightarrow T + F * F$
 $\Rightarrow T + \text{num} * F$
 $\Rightarrow F + \text{num} * F$
 $\Rightarrow \text{id} + \text{num} * F$
 $\Rightarrow \text{id} + \text{num} * \text{id}$

which matches the input sequence $\text{id}(x) + \text{num}(2) * \text{id}(y)$. Top down parsing occasionally requires backtracking. For example, suppose we used the derivation $E \Rightarrow E - T$ instead of the first derivation. Then, later we would have to backtrack because the derived symbols will not match the input tokens. This is true for all nonterminals that have more than one production since it indicates that there is a choice of which production to use. We will learn how to construct parsers for many types of CFGs that never backtrack. These parsers are based on a method called *predictive parsing*. One issue to consider is which nonterminal to replace when there is a choice. For example, in $T + F * F$ we have 3 choices: we can use a derivation for T , for the first F , or for the second F . When we always replace the leftmost nonterminal, it is called *leftmost derivation*.

In contrast to top-down parsing, *bottom-up parsing* starts from the input string and uses derivations in the opposite directions (ie, by replacing the rhs sequence $X_1X_2...X_n$ of a production $A ::= X_1X_2...X_n$ with the nonterminal A . It stops when it derives the start symbol. For example,

$\text{id}(x) + \text{num}(2) * \text{id}(y)$
 $\Rightarrow \text{id}(x) + \text{num}(2) * F$
 $\Rightarrow \text{id}(x) + F * F$
 $\Rightarrow \text{id}(x) + T * F$
 $\Rightarrow \text{id}(x) + T$
 $\Rightarrow F + T$
 $\Rightarrow T + T$
 $\Rightarrow E + T$
 $\Rightarrow E$

The *parse tree* of an input sequence according to a CFG is the tree of derivations. For example if we used a production $A ::= X_1X_2...X_n$ (in either top-down or bottom-up parsing) then we construct a tree with node A and children $X_1X_2...X_n$. For example, the parse tree of $\text{id}(x) + \text{num}(2) * \text{id}(y)$ is:



F	F	id
id	num	

So a parse tree has non-terminals for internal nodes and terminals for leaves. As another example, consider the following grammar:

```

S ::= ( L )
    | a
L ::= L , S
    | S

```

Under this grammar, the parse tree of the sentence $(a,((a, a), a))$ is:

```

      S
    / | \
  (  L  )
  / | \
L   ,   S
|   / | \
S  (  L  )
|   / | \
a  L   ,   S
    |       |
    S       a
  / | \
(  L  )
 / | \
L   ,   S
|   |   |
S   a
|
a

```

Consider now the following grammar G2:

```

E ::= T + E
    | T - E
    | T
T ::= F * T
    | F / T
    | F
F ::= num
    | id

```

This is similar to our original grammar, but it is right associative when the leftmost derivation rules is used. That is, $x-y-z$ is equivalent to $x-(y-z)$ under G2, as we can see from its parse tree.

Consider now the following grammar G3:

```

E ::= E + E
    | E - E
    | E * E
    | E / E
    | num
    | id

```

Is this grammar equivalent to our original grammar G1? Well, it recognizes the same language, but it constructs the wrong parse trees. For example, the $x+y*z$ is interpreted as $(x+y)*z$ by this grammar (if we use leftmost derivations) and as $x+(y*z)$ by G1 or G2. That is, both G1 and G2 grammar handle the operator precedence correctly (since $*$ has higher precedence than $+$), while the G3 grammar does not.

In general, to write a grammar that handles precedence properly, we can start with a grammar that does not handle precedence at all, such as our last grammar G3, and then we can refine it by creating more nonterminals, one for each group of operators that have the same precedence. For example, suppose we want to parse an expression E and we have 4 groups of operators: $\{not\}$, $\{*, /\}$, $\{+, -\}$, and $\{and, or\}$, in this order of precedence. Then we create 4 new nonterminals: N , T , F , and B and we split the derivations for E into 5 groups of derivations (the same way we split the rules for E in the last grammar into 3 groups in the first grammar).

A grammar is *ambiguous* if it has more than one parse tree for the same input sequence depending which derivations are applied each time. For example, the grammar G3 is ambiguous since it has two parse trees for $x-y-z$ (one parses $x-y$ first, while the other parses $y-z$ first). Of course, the first one is the right interpretation since $-$ is left associative. Fortunately, if we always apply the leftmost derivation rule, we will never derive the second parse tree. So in this case the leftmost derivation rule removes the ambiguity.

Next	Up	Previous	Contents
------	----	----------	----------

Next: [3.2 Predictive Parsing](#) **Up:** [3 Parsing](#) **Previous:** [3 Parsing](#) [Contents](#)
fegaras 2012-01-10