



Compiler Construction

Lecture Notes

Syntax Analysis

Agenda

Top-Down Parsing

Bottom-Up Parsing

The general form of top-down parsing, called recursive-descent parsing, may require backtracking to find the correct production to be applied. Predictive parsing is a special case of recursive-descent parsing, not requiring backtracking. Predictive parsing chooses the correct production by looking ahead at the input a fixed number of symbols, typically only one (the next input symbol).

Top-down parse tree of the expression above constructs a tree with two nodes labeled E' . At the first E' node (in preorder), the production $E' \rightarrow +TE'$ is chosen; at the second E' node, the production $E' \rightarrow \text{£}$ is chosen. A predictive parser can choose between E' -productions by looking at the next input symbol.

The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is called the $LL(k)$ class.

Recursive-Descent Parsing:

A typical procedure for a nonterminal in a top-down parser is shown below:

```
void A() {
    Choose an A-production,  $A \rightarrow X_1X_2, \dots, X_k$ ;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts successfully if its procedure body scans the entire input string.

A recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not common.

To allow backtracking, the pseudo code above needs to be modified. First, it cannot choose a unique A-production at line (1), so try each of several productions in some order. Then, failure at line (7) of the code suggests to return to line (1) and try another A-production.

Only if there are no more A-productions to try, declare that an input error has been found. In order to try another A-production, reset the input pointer to where it was when we first reached line (1). Thus, a local variable is needed to store this input pointer for future use.

Example : Construct top-down parse tree for the grammar below:

$$\begin{aligned} S &\rightarrow c A d \\ A &\rightarrow a b \mid a \end{aligned}$$

To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labeled S , and the input pointer pointing to c , the first symbol of w . S has only one production, so we use it to expand S and obtain the tree shown below. The leftmost leaf, labeled c , matches the first symbol of input w , so we advance the input pointer to a , the second symbol of w , and consider the next leaf, labeled A .

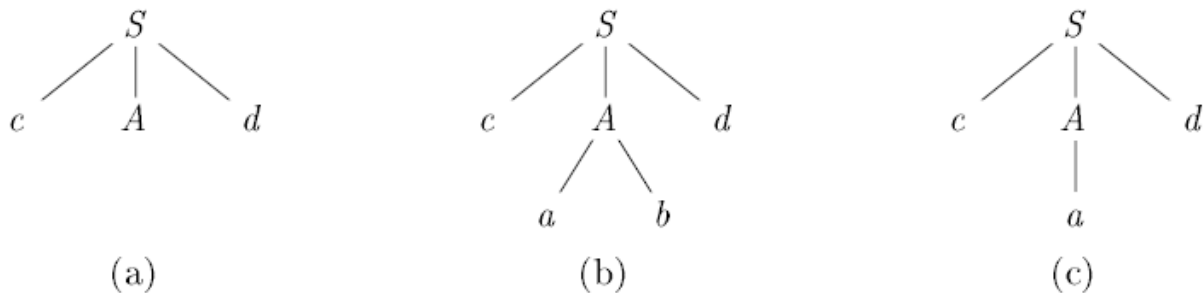


Figure 2: Steps of top-down parse

Expand A using the first alternative $A \rightarrow ab$ to obtain the above tree. We have a match for the second input symbol, a , so we advance the input pointer to d , the third input symbol, and compare d against the next leaf, labeled b . Since b does not match d , we report failure and go back to A to see whether there is another alternative for A that has not been tried, but that might produce a match.

In going back to A , reset the input pointer to position 2, the position it had when we first came to A , which means that the procedure for A must store the input pointer in a local variable.

The second alternative for A produces the tree of Fig2 (c). The leaf a matches the second symbol of w and the leaf d matches the third symbol.

Since we have produced a parse tree for w , we halt and announce successful completion of parsing.

LL(1) Grammars:

Predictive parsers or recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first “L” in LL(1) stands for scanning the input from left to right, the second “L” for producing a leftmost derivation, and the “1” for using one input symbol of lookahead at each step to make parsing action decisions.

The class of LL(1) grammars can describe most programming constructs, but there are important points to consider when writing a suitable grammar for the source language. For example, left-recursive or ambiguous grammar cannot be LL(1).

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G , the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a .
2. At most one of α and β can derive the empty string.

Predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at one input symbol. Flow-of-control constructs, with their distinguishing keywords, generally satisfy the LL(1) constraints. For instance, if we have the productions:

$$\begin{array}{lcl} stmt & \rightarrow & \text{if (} expr \text{) } stmt \text{ else } stmt \\ & & | \quad \text{while (} expr \text{) } stmt \\ & & | \quad \{ stmt_list \} \end{array}$$

then the keywords `if`, `while`, and the symbol `{` tell us which alternative is the only one that could possibly succeed parsing.

Bottom-Up Parsing

It is convenient to describe parsing as the process of building parse trees, but the front end usually carries out translation directly without building an explicit tree.

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

For the expression grammar above, Figure 3 illustrates a bottom-up parse of the token stream `id * id`, with respect to the expression grammar.

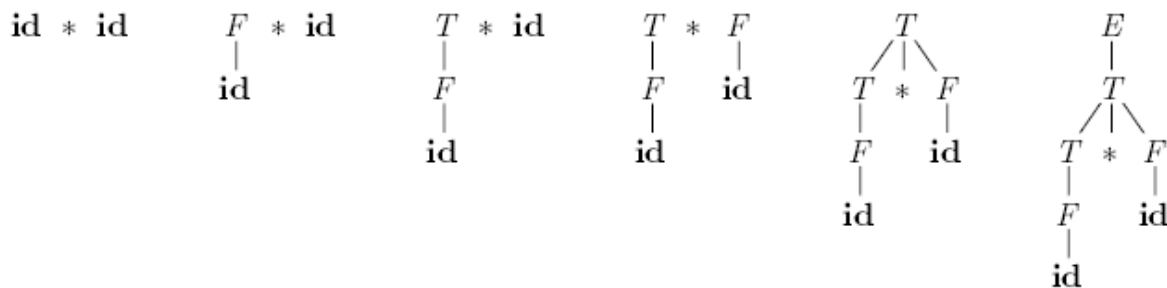


Figure 3: A bottom-up parse for `id * id`