Top-Down Parsing

Parsing is the process of determining how a string of terminals can be generated by a grammar. In discussing this problem, it is helpful to think of a parse tree being constructed, even though a compiler may not construct one, in practice. However, a parser must be capable of constructing the tree in principle, or else the translation cannot be guaranteed correct.

Possible Approaches

The syntax analysis phase of a compiler verifies that the sequence of tokens extracted by the scanner represents a valid sentence in the grammar of the programming language. There are two major parsing approaches: top¬down and bottom¬up. In top-down parsing, you start with the start symbol and apply the productions until you arrive at the desired string. In bottom-up parsing, you start with the string and reduce it to the start symbol by applying the productions backwards. As an example, let's trace through the two approaches on this simple grammar that recognizes strings consisting of any number of a's followed by at least one (and possibly more) b's:

S –> AB
A –> aA | ε
B –> b | bB

Here is a top¬down parse of aaab. We begin with the start symbol and at each step, expand one of the remaining nonterminals by replacing it with the right side of one of its productions. We repeat until only terminals remain. The top-down parse produces a leftmost derivation of the sentence.

S
AB S –> AB
aAB A –> aA
aaAB A –> aA
aaaAB A –> aA
aaaεB A –> ε
aaab B –> b

A bottom¬up parse works in reverse. We begin with the sentence of terminals and each step applies a production in reverse, replacing a substring that matches the right side with the nonterminal on the left. We continue until we have substituted our way back to the start symbol. If you read from the bottom to top, the bottom-up parse prints out a rightmost derivation of the sentence.

aaab
aaaεb (insert ε)
aaaAb A –> ε
aaAb A –> aA
aAb A –> aA
Ab A –> aA

AB B –> b

S S –> AB

In creating a parser for a compiler, we normally have to place some restrictions on how we process the input. In the above example, it was easy for us to see which productions were appropriate because we could see the entire string aaab. In a compiler's parser, however, we don't have long¬distance vision. We are usually limited to just one¬symbol of lookahead. The lookahead symbol is the next symbol coming up in the input. This restriction certainly makes the parsing more challenging. Using the same grammar from above, if the parser sees only a single b in the input and it cannot lookahead any further than the symbol we are on, it can't know whether to use the production B–> b or B –> bB.

Backtracking

One solution to parsing would be to implement backtracking. Based on the information the parser currently has about the input, a decision is made to go with one particular production. If this choice leads to a dead end, the parser would have to backtrack to that decision point, moving backwards through the input, and start again making a different choice and so on until it either found the production that was the appropriate one or ran out of choices. For example, consider this simple grammar:

S –> bab|bA

A –> d|cA

Let's follow parsing the input bcd. In the trace below, the column on the left will be the expansion thus far, the middle is the remaining input, and the right is the action attempted at each step:

S bcd Try S –> bab
Bab bcd match b
ab cd dead-end, backtrack
S bcd Try S –> bA
bA bcd match b
A cd Try A –> d
d cd dead-end, backtrack
A cd Try A –> cA
cA cd match c
A d Try A –> d
d d match d

Success!

As you can see, each time we hit a dead-end, we backup to the last decision point, unmake that decision and try another alternative. If all alternatives have been exhausted, we back up to the preceding decision point and so on. This continues until we either find a working parse or have exhaustively tried all combinations without success. Backtracking parsers canbeused for a variety of grammars without requiring them to fit any specific form. For a small grammar such as above, a backtracking approach may be tractable, but most programming language grammars have dozens of nonterminals each with several options and the resulting combinatorial explosion makes a this approach very slow and impractical. We will instead look at ways to parse via efficient methods that have restrictions about the form of the grammar, but usually those requirements are not so onerous that we cannot rearrange a programming language grammar to meet them.

Top¬Down Predictive Parsing

First, we will focus in on top¬down parsing. We will look at two different ways to implement a non¬backtracking

top¬down parser called a predictive parser. A predictive parser is characterized by its ability to choose the production to apply solely on the basis of the next input symbol and the current nonterminal being processed. To enable this, the grammar must take a particular form. We call such a grammar LL(1). The first "L" means we scan the input from left to right; the second "L" means we create a leftmost derivation; and the 1 means one input symbol of lookahead. Informally, an LL(1) has no left-recursive productions and has been left¬factored. Note that these are necessary conditions for LL(1) but not sufficient, i.e., there exist grammars with no left¬recursion or common prefixes that are not LL(1). Note also that there exist many grammars that cannot be modified to become LL(1). In such cases, another parsing technique must be employed, or special rules must be embedded into the predictive parser.

Recursive Descent

The first technique for implementing a predictive parser is called recursive¬descent. A recursive-descent parser consists of several small functions, one for each nonterminal in the grammar. As we parse a sentence, we call the functions that correspond to the left side nonterminal of the productions we are applying. If these productions are recursive, we end up calling the functions recursively.

Algorithm for Recursive descent parsing

void A() {
1) Choose an A-production, A -> XI X2 . . . Xk;
2) for (i = l t o k) {
3 if (Xi is a nonterminal)
4) call procedure Xi () ;
5 else if (Xi equals the current input symbol a)
6) advance the input to the next symbol;
7) else /* an error has occurred */;
}
}

Let’s start by examining some productions from a grammar for a simple Pascal¬like programming language. In this programming language, all functions are preceded by
the reserved word FUNC: program –> function_list function_list –> function_list function | function function –> FUNC identifier (parameter_list) statements
.

What might the C function that is responsible for parsing a function definition look like? It expects to first find the token FUNC, then it expects an identifier (the name of the function), followed by an opening parenthesis, and so on. As it pulls each token from the parser, it must ensure that it matches the expected, and if not, will halt with an error. For each nonterminal, this function calls the associated function to handle its part of the parsing. Check this out:

```
void ParseFunction() {
if (lookahead != T_FUNC) { // anything not FUNC here is wrong printf("syntax error \n"); exit(0);
} else
lookahead = yylex();     // global 'lookahead' holds next token ParseIdentifier(); if (lookahead != T_LPAREN) {
printf("syntax error \n"); exit(0); } else
lookahead = yylex(); ParseParameterList(); if (lookahead!= T_RPAREN) {
```

printf("syntax error \n"); exit(0); } else lookahead = yylex(); ParseStatements(); }

To make things a little cleaner, let's introduce a utility function that can be used to verify that the next token is what is expected and will error and exit otherwise. We will need this again and again in writing the parsing routines.
void MatchToken(int expected) {
if (lookahead != expected) { printf("syntax error, expected %d, got %d\n", expected,lookahead); exit(0);
} else // if match, consume token and move on lookahead = yylex(); } Now we can tidy up the ParseFunction routine and make it clearer what it does:
void ParseFunction()
{ MatchToken(T_FUNC); ParseIdentifier(); MatchToken(T_LPAREN); ParseParameterList();
MatchToken(T_RPAREN); ParseStatements();
}
Here is the production for an if¬statement in this language:
To prepare this grammar for recursive¬descent, we must left¬factor to share the common parts:
if_statement –> IF expression THEN statement close_ifclose_if –> ENDIF | ELSE statement ENDIF
Now, let's look at the recursive¬descent functions to parse an if statement:
void ParseIfStatement()
{ MatchToken(T_IF); ParseExpression(); MatchToken(T_THEN); ParseStatement(); ParseCloseIf();
}
void ParseCloseIf() { if (lookahead == T_ENDIF) // if we immediately find ENDIF lookahead = yylex(); // predict close_if -> ENDIF
else { MatchToken(T_ELSE); // otherwise we look for ELSE ParseStatement(); // predict close_if -> ELSE stmt ENDIF MatchToken(T_ENDIF);
} }
When parsing the closing portion of the if, we have to decide which of the two right-hand side options to expand. In this case, it isn't too difficult. We try to match the first token again ENDIF and on non¬match, we try to match the ELSE clause and if that doesn't match, it will report an error.
Navigating through two choices seemed simple enough, however, what happens where we have many alternatives on the right side?
statement –> assg_statement | return_statement | print_statement | null_statement| if_statement | while_statement | block_of_statements
When implementing the ParseStatement function, how are we going to be able to determine which of the seven options to match for any given input? Remember, we are trying to do this without backtracking, and just one token of lookahead, so we have to be able to make immediate decision with minimal information— this can be a challenge.

Left Recursion

What about left-recursive productions? Now we see why these are such a problem in a predictive parser. Consider this left¬recursive production that matches a list of one or more functions.
function_list –> function_list function | function function –> FUNC identifier (parameter_list) statement
void ParseFunctionList()

{ ParseFunctionList(); ParseFunction();
}
Such a production will send a recursive¬descent parser into an infinite loop! We need to remove the left¬recursion in order to be able to write the parsing function for a function_list.
function_list –> function_list function | function
becomes
function_list –> function function_list | function
then we must left¬factor the common parts
function_list –> function more_functions more_functions –> function more_functions | ε
And now the parsing function looks like this:
void ParseFunctionList()
{
ParseFunction();
ParseMoreFunctions(); // may be empty (i.e. expand to epsilon)
}

Elimination of Left Recursion

A grammar is left recursive if it has a nonterminal A such that there is a derivation A –>Aa for some string a. Top-down parsing methods cannot
handle left-recursive grammars, so a transformation is needed to eliminate left
recursion.

Algorithm : Eliminating left recursion.

INPUT: Grammar G with no cycles or e-productions.
OUTPUT: An equivalent grammar with no left recursion.
METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting
non-left-recursive grammar may have E-productions.
1) arrange the nonterminals in some order A1, A2, . . . , A,.
2) for (each i from 1 to n) {
3) for (each j from 1 to i - 1) {
4) replace each production of the form Ai -> AjB by the
productions Ai □C1B I C2B |.. . I CkB, where
Aj -> Cl | C2 | . . . | Ck are all current Aj-productions
}
6) eliminate the immediate left recursion among the Ai-productions
7) }