**A Recursive Descent Parser**

**Ramon A. Mata-Toledo Ph.D.**

**Rollins College**

**Saturday, October 23, 2021**

The one-push down recognizer covered in class using the SELECT sets showed how LL(1) grammars can be recognized in a top-down derivation implemented with a pushdown machine [1,4]. In this document, we present an alternative, but related, method of recognizing and translating LL(1) grammars. This alternative method is known as the **recursive descent** and is suitable when the programming language of the compiler allows recursive procedures. (A procedure is *recursive* if it can call on itself, either directly or indirectly through a chain of other procedure calls.). Although most of the compiler courses taught nowadays only concentrate on tools, even the tools do recursive descent under their hood, so it is good to know this technique.

**A Recursive Descent Parser**

The basic idea of recursive descent is that each nonterminal of the grammar has a corresponding procedure which recognizes an example of that nonterminal. These procedures call on each other when appropriate. The stacking mechanism of the pushdown machine is thus supplanted by the procedure-calling mechanism of the high-level language. Using the method of recursive descent, a syntax analyzer can be written as fast as one can write. To illustrate this procedure, we will use a simple grammar as the one shown below. *The nonterminal symbols are written in uppercase and the terminal in lowercase*. The grammar is defined by the productions indicated below.

**Note to the reader**: This professor has used this grammar as an example of the recursive method before but, unfortunately, he does not remember how he got possession of it or who the author is. Therefore, I cannot give you the reference nor I claim ownership of the example. Nevertheless, the explanations, comments, and hints are all mine.

**Example No. 1** (initial grammar) (2)

PROGRAM -> a begin DECLIST comma STATELIST end

DECLIST -> d semi DECLIST

DECLIST -> d

STATELIST -> s semi STATELIST

STATELIST -> s

This grammar is not LL(1) and is not suitable for writing a compiler for it because it needs to be left-factorized [1.4]. In case, the reader does not remember why this is a problem a revision of the class notes is in order.

Following the rules of removal of left factorization, the given grammar can be transformed as indicated next. Notice that there are two new nonterminal symbols X and Y after removing the productions that have to be left-factorized. This transformed grammar is the one we use to write the recursive descent syntax analyzer or parser. The word “epsilon” stands for the “empty string.”

PRGRAM -> begin DECLIST comma STATELIST end

DECLIST -> d X

X -> semi DECLIST

X -> epsilon

STATELIST -> s Y

STATELIST -> s Y

Y -> semi STATELIST

Y -> epsilon

**The remaining of this page has been left blank on purpose**

The next step is to write a procedure to recognize each nonterminal symbol. The syntax does not follow any of the current languages, but it should be clear what it is intended with each procedure.

**Conventions used in this program**

1. The beginning and ending of the body of each function start and end with a opening and closing curly brackets respectively. After the name of every function there must be a pair of parentheses followed by a “:”. Statements within the function are indented at least one space and enclosed in the curly brackets.
2. Conditional statements are of the form: **if**…**then**. Each conditional end with **fi** # Someone thought that if should end in “fi”
3. **elif º else if. # The three bars means “equivalent to”**
4. The assignment statement is indicated by the compound symbol “:=”

**function** **PROGRAM**(): # beginning of recursive program

**{**

**if** symbol ≠ begin **then** error **fi**

symbol := lexical # This procedure reads the next token.

DECLIST

**if** symbol ≠ comma **then** error **fi**

symbol := lexical

STATELIST

**if** symbol ≠ end **then** error **fi**

**}** # end of function program

**function DECLIST**():

**{**

**if** symbol ≠ d **then** error **fi**

symbol := lexical

X

**}** # end of function DECLIST

**function** **X**()**:**

**{**

**if** symbol = semi **then**

symbol := lexical DECLIST

**elif** symbol = comma

**then skip**

**else** error

**fi**;

**}** # end of function X

**function STATELIST**():

{

**if** symbol ≠ s **then** error **fi**

symbol:= lexical

Y

} # end of function STATELIST

**function Y**():

{

**if** symbol = semi **then**

symbol := lexical STATELIST

**elif** symbol = end **then skip**

**else** error

**fi**} # end of function Y

**Additional Conventions:**

The lexical designer must define a “good neighbor” protocol. The designer must determine if the symbol that is going to be the current token to be used by a function is passed to the function (as a courtesy) or if the function has to call the lexical analyzer as the first statement in its body. The good neighbor” policy used here is to pass the current token to the next procedure by calling lexical who stores the current token in a global variable. It is the decent thing to do. “It’s valid.”

The identifiers semi and comma are tokens whose values are the compile-time representation of the terminal symbols by the same name of the grammar.

Following the “good neighbor” policy it is assumed that before the function PROGRAM is called the current token has been placed in the global variable by the lexical analyzer. However, throughout the program, there is a master-slave relationship between the lexical analyzer (the slave) and the syntax analyzer (the master). That is, the function of the lexical analyzer is to find and return the next token (current symbol) to the syntax analyzer.

**Advantages and Disadvantages of writing a recursive descent parser**

The main advantage is the speed with which the parser can be written. Notice that we only need to just follows the productions of the language augmented with the FIRST and FOLLOW SET. Because the close correspondence between the grammar and the parser, once the grammar has been transformed properly, there is a high probability that the parser be correct or that its errors, if any are minimal. In addition, through the use of the FIRST and FOLLOW sets error recovery can be incorporated directly into a Recursive Descent parser.

The downsize of the recursive descent parser is that it is “slow” due to the high incidence of functions calls it tends to make.

**Note No. 2**

The function **error**() is designed to deal with errors, although the details are not considered in the example, the reader should read pages 11through 18 (exclude the section “output” on page 18). Pay particular attention to the three general cases on page 15 and the Error Recovery Rules 1 and 2 listed on page 13. Notice the keys contain all symbols from which compilation can resume (FOLLOW set). The Handles include the key words of the language plus the FOLLOW and FIRST.

**Hint**: As indicated by Hartmann on page 17, “before programming a parser in this scheme (the recursive descent) one must master the three general cases on page 15.” Therefore, pay attention to the example on page 17 where he writes the corresponding code to the “if statement” shown on page 87 (number 27) and the example of “term” on page 18 of diagram of term shown on page 88 (number 37). If you understand these “three basics” constructs and how they are applied to the programming of the parser, you can write any lexical analyzer that comes your way. That is, you are “valid.”

*My personal advice is that you try to do on your own first the “if” and “term” examples before mentioned and then then check what you did against Hartmann’s solution*. If you are right on the money, then you have learned how to write a lexical analyzer. As you guys say, “No cap”.

**References**

1. Compiler Design by Lewis II, Stearns, Rosenkratz, The IBM System Programming Series. Addison-Wesley. 1972.
2. Unknown author at least to this professor.
3. A Concurrent Pascal Compiler for Minicomputers by A. Hartmann. Springer-Verlang. 1980
4. The Theory of Parsing, Translation, and Compiling by A. Aho and J. Ulmann. Prentice-Hall. 1972.