Implementation of Learning for Syntax Analyzer

Hnin Myat Soe, Daw Mar Mar Lwin
University of Computer Studies, Yangon
wintergirl86@gmail.com, dawmarmarlwin@gmail.com

Abstract

Learning Compiler is the foundation of any programming language implementations. Learning tool provides students to understand the compiling techniques in effective ways. This paper defines effective learning tool of a Top-Down Parsing Technique by using Java Language. To implement the procedures of how top down parser works, it needs to analyze the internal processing for checking input source token formats Parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language and produces as output a parse tree to use for semantic analysis. C++ grammar is used LL (1) (left-to-right, leftmost derivation). This paper explains how to construct the Non-Recursive (Table Driven) Predictive Parser that is also known as LL(1) parser and how to implement the parser as a learning tool.
Keywords: Compiler, Top Down Parsing Technique, LL(1), Non-Recursive Predictive Parser

1. Introduction

Software may be written in a wide variety of languages. A language compiler is one of the most crucial tools for the software writing development. Compilers and Interpreters are a necessary part of any computer system—without them, there will be programmed by assembly language or machine language.

A well written compiler is highly modular in design. The compilation process is essentially a transformation from one language (source code) to another (object code). Logically, the compilation process is divided into stages, which in turn divided into phases. Physically, the compiler is divided into passes. The principal stages that are presented in compiler are analysis and synthesis. In the analysis stage, the source code is analyzed to determine its structure and meaning. Synthesis in which object codes is built or synthesized. The analysis stage is usually assumed to consist of three distinct phases, Lexical analysis, Syntax analysis and semantic analysis.

The first phase, called lexical analyzer or scanner, separates characters of the source language into group that logically belong together, these groups are called tokens. The output of lexical analyzer is a stream of tokens, which is passed to the next phase, the syntax analyzer, or parser. The syntax analyzer groups tokens together into syntactic structure. The last phase Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically proceeds the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation [9].

2. Related works

The two dominant parsing techniques in real compilers are LL(1) and LALR(1). Because the underlying algorithms are more complicated, most LALR(1) parsers are built using parser generators such as *yacc* and *bison*. LL(1) parsers may be implemented via hand-coded recursive-descent or via LL(1) table-driven predictive parser generators like *LLgen* [1].

Parsing methods became much more systematic as the theoretical work progressed. Several general techniques for parsing any CFG (Context Free Grammar) were invented in the 1960's. LR grammars and parsers were first introduced by Knuth [2] who described the construction of LR parsing tables. [4] was the first to show parsers for programming languages could be produced using these techniques. DeRemer[5][6] devised the more practical SLR and LALR techniques. This laid the groundwork for automatic parser generators. YACC was built by S.C. Johnson in the early 1970's. Today, most compilers are created using automatic parser generators. The idea of LL(1) grammars was introduced by Lewis and Stearns [7]. Predictive parsers were first discussed by Knuth [3]. Recursive-descent and table-driven parsing techniques were shown to be useful for programming languages by Lewis, Rosenkrantz and Stearns [8]. These

techniques were widely used in early compilers but nowadays bottom-up seems more dominant.

3. Overview of the parser

Logically, the compilation process is divided into stages, which in turn divided into phases. Physically, the compiler is divided into passes. The principal stages that are presented in compiler are analysis and synthesis. In the analysis stage, the source code is analyzed to determine its structure and meaning. Synthesis in which object codes is built or synthesized. The analysis stage is usually assumed to consist of three distinct phases, syntax analysis, Lexical analysis and semantic analysis.

Syntax analysis or parsing is constructing a derivation of a sentence according to the rules of a specified grammar. A syntax analyzer or parser is device or procedure for performing syntax analysis. Parsing is the process for determining if a string of token can be generated by a grammar. Grammars used in practice, have a special form. Programming language parsers almost always make a single left to right scan over the input, looking ahead one token at a time.

The operation if two basic types of parsers for the grammar are top-down and bottom-up. Top down parsing method build parse tree from the top (root) and work down to the bottom (leaves) while bottom up parsing method build parse tree from the bottom (leaves) to the top.

During the parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code.

3.1. Top-down parsing techniques

There are two types of Top-Down Parsing Techniques; they are Recursive-Descent Parsing and Predictive Parsing. Recursive-Decent Parsing, needs backtracking (If a choice of a production rule does not work, we backtrack to try other alternatives.). It is a general parsing technique, but not widely used and it is not an efficient technique

Predictive Parsing, is a special form of Recursive Decent Parsing, it does not need backtracking and it is an efficient technique. It needs a special form of grammars (LL(1) grammars). Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

3.2. Syntax definition or grammar

A grammar is used to generate sequence of symbols that make up the sentences of a language by stating with the sentence symbol and successively replacing it or a non-terminal in a sentence derived form it, using one of the productions of the grammar. A grammar is defined to be a quadruple.

(V_T, V_N, P, S)

Where,

V_T is an alphabet whose symbols are known as terminal symbols.

V_N is an alphabet knows as non terminal symbols.

V_N and V_T have no symbol in common.

P is the set of productions (rules).

S is the member of V_N and is known as the staring symbol of the grammar and is the starting symbol if the grammar and is the starting point in the generation of any sentence in the language.

3.3. Derivations

The rules of a grammar are used to define a special kind of string substitution. This substitution is accomplished by replacing a specific non-terminal in some given string of terminals and non-terminals with the right-hand side of a production, which has the specified non-terminal as its left had side.

For example, consider the following grammar with starting non-terminal. <S>

1. <S> → <X> <Y>
2. <S> → ε
3. <X> → x <S> <X>
4. <Y> → y <Y>
5. <X> → x
6. <Y> → y

If we are given the sentence x x x y y, the leftmost derivation will get from given grammar.

<S> => <X> <Y> => x <S> <X> <Y>
=> x <X> <Y>
=> x x <S> <X> <Y> => x x <X>
<Y> => x x x <Y>
=> x x x y <Y> => x x x y y

3.4. Predictive parser

In many cases, by carefully writing grammar, eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive-descent parser that needs no backtracking, i.e., a predictive parser.

To construct predictive parser, we must know, given the current input symbol *a* and the non-terminal *A* to be expanded, which of the alternatives of production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ is the unique alternative that derives a string beginning with *a*. That is, the proper alternatives must be detectable by

looking at only the first symbol it derives. Flow-of control constructs in most programming languages, with their distinguishing keywords, are usually detectable in this way. For example, in C++ if we have the productions,

stmt → if *expr* then *stmt* else *stmt*
| while *expr* stmt
| do *stmt* while (*expr*) ;

Then the keyword if, while, and do tell us which alternative is the only one that could possibly succeed if we are to find a statement.

Grammars whose predictive parsing tables contain no duplicate entries are called LL(1). This stands for *left-to-right parse, leftmost-derivation, 1-symbol lookahead*.

Clearly a recursive-descent (predictive) parser examines the input left-to-right in one pass (some parsing algorithms do not, but these are generally not useful for compilers). The order in which a predictive parser expands non-terminals into right-hand sides (that is, the recursive-descent parser calls functions corresponding to non-terminals) is just the order in which a leftmost derivation expands non-terminals. And a recursive-descent parser does its job just by looking at the next token of the input, never looking more than one token ahead.

3.4.1. First-order headings

Consider the following grammar for arithmetic expressions.

Grammar (2.1)

E → E + T | T
T → T * F | F
F → (E) | id

Eliminating the immediate left recursion to the productions for E and then for T, we obtain

Grammar (2.2)

E → T E '
E ' → + T E ' | ε
T → F T'
T ' → * F T ' | ε
F → (E) | id

We can eliminate immediate left recursion from them by following technique. First, we group the X productions as X→ X α_1 | X α_2| … | X α_m | β_1 | β_2 | … | β_n

Where no β_1 begins with an X. Then, we replace the X productions by

X → β_1X' | β_2X' | …β_nX'
X' → α_1 X' | α_2 X' |… | α_m X'

The non terminal X generates the same strings as before but is no longer left recursive. This eliminates left recursion from a grammar. It is guaranteed to work if the grammar has no cycles or ε productions.

3.4.2. Elimination of Left Recursion Algorithm

Input Grammar G with no cycle or ε productions.
Output An equivalent grammar with no left recursion
Method

The resulting non-left recursive grammar may have ε productions

1. Arrange the non-terminals in some order $X_1, X_2, \ldots, X_n$
2. For j= 1 to n do begin
   For k=1 to j-1 do begin
   Replace each production of the form $X_j \rightarrow X_j \theta$ by the productions
   Xj → $\delta_1\theta$ | $\delta_2\theta$ | … | $\delta_i\theta$
   Where $X_k \rightarrow \delta_1 | \delta_2 | \ldots | \delta_i$ are all the current X_k productions
   End
   Eliminate the immediate left recursion among the X_j productions;
   End

The reason the procedure works that after the j - 1st iteration of the outer for loop in step (2), any production of the form $X_i \rightarrow X_1 \alpha$, where I <j , must have i > 1. As a result, on the next iteration, the inner loop (on k) progressively raises the lower limit on n in any production $X_j \rightarrow X_i \alpha$, until we must have n>j. Then, eliminating immediate left recursion for the X_j productions forces i to be greater than j.

4. Implementation for non-recursive predictive parser

It is possible to build a non recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a non terminal. The non recursive parser in Figure 1 looks up the production to be applied in a parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.

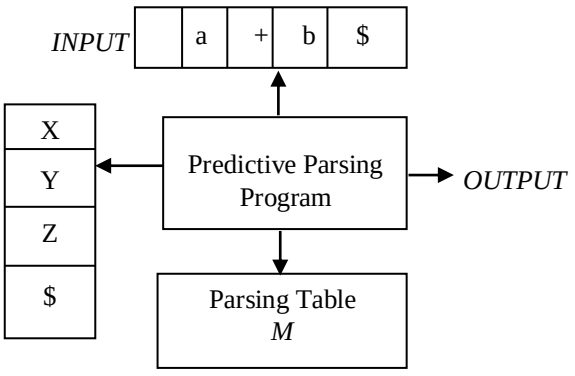

Figure 1. Model of a non recursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack.

Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M [A , a], where A is a non terminal and a is terminal symbol or the symbol $.

The parser is controlled by a program that behaves as follows.

The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non terminal, the program consults entry M [X, a] of the parsing table M. This entry will be either an X-production of the grammar of an error entry. If, for example, M [X, a] = {$X \rightarrow UVW$ }, the parser replaces X on top of the stack by WVU (with U on the top). As output, we shall assume that the parser just prints the productions used; any other code could be executed here. If M [X, a] = error, the parser calls an error recovery routine.

4.1. System Design

The behavior of the parser can be described in terms of its configurations, which give the stack contents and the remaining input. The system design overview can be seen in Figure 2.

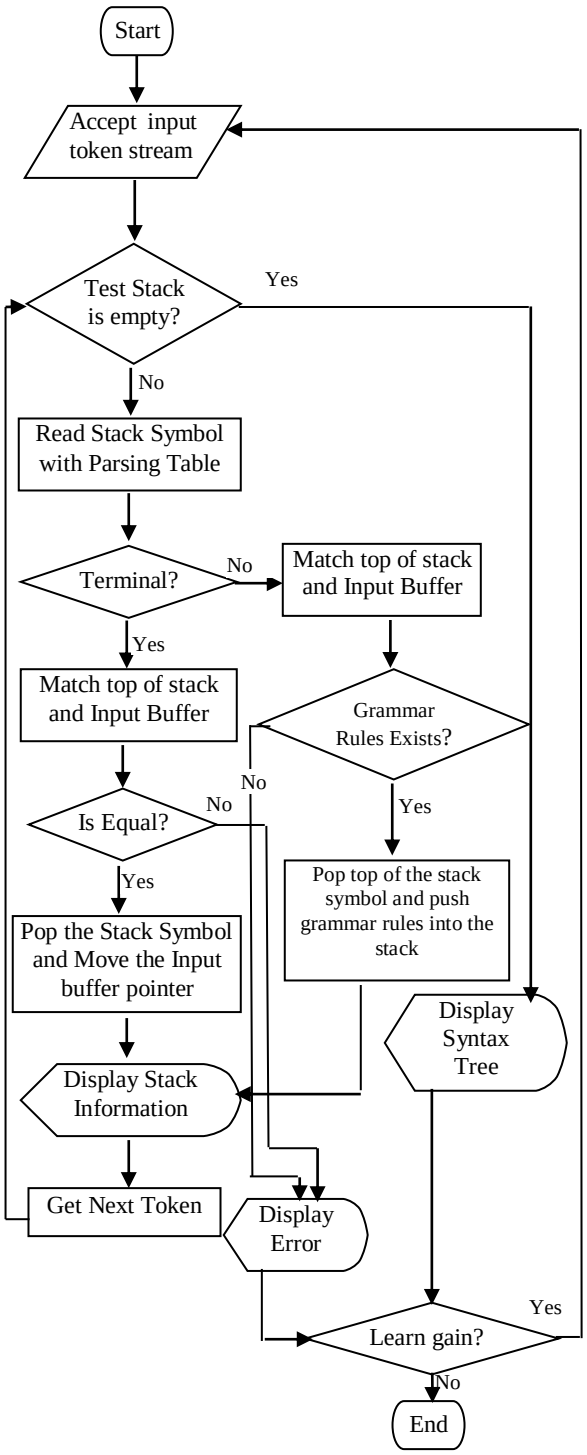

Figure 2. System flowchart of non predictive parser learning and testing tool

The predictive parsing algorithm is as shown in below.

Set *ip* to point to the symbol of *w*$:

Repeat

Let X be the top stack symbol and a the symbol pointed to by *ip*;

If X is a terminal or $ then
If $X = a$ then
Pop X from the stack and advance *ip*
Else error( )
Else /* X is a non terminal */
If $M [X, a] = X \rightarrow Y_1, Y_2, \ldots Y_k$ then
Begin
Pop X from the stack;
Push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y1 on top;
Output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$
End
Else error( )
Until X = $ /* Stack is empty */

4.2 Building learns and test modules

This system is built learn and test modules for the Top down Parsing Technique. The Top Down Parsing Techniques can be learn from this system's learning dialogs and for testing the user's input token stream, this system accepts the input tokens of the parser (See Figure 2). These input token streams are work with the top down parsing algorithm. The procedures for showing success or error messages, syntax tree messages, parsed grammar rules and stack information can be viewed for understanding the parsing stages. This system dynamically captures the results of the parsing process with String Builder message store and it is retrieved by the system according to the error status or success status.

4.3 Building grammar rules and parse tables

By applying the elimination of left recursion algorithm, we build the C++ grammar rules as follows:
Grammar 1 shows the grammar for C++ sample input token stream

Grammar 1.
1. <program> → <header_list> VOID MAIN() <prog_block>
2. <header_list> → <header_list><header>| <header>
3. <header> → #INCLUDE <HEADER_FILE>
4. <prog_block> → { <block_stmt> }
5. <block_stmt> → <dec_list> <stmt_list>| <stmt_list>
6. <dec_list> → <type> <id_list> ;
7. <type> → INT | FLOAT | CHAR
8. <id_list> → <id_list> ID| ID
9. <stmt_list> → <stmt_list><stmt>| <stmt>
10. <stmt> → <assign>| <write> | <read>| <if>| <for> | <while>

…
Grammar 2 shows the grammar of modified elimination of left recursion algorithm from *grammar 1*. (This system uses 53 grammar rules for testing input parser token streams and 81 parsing tables identifiers are generated from these grammar rules as shown in below.)

Grammar 2:
R1. <program> → <header_list> VOID MAIN() <prog_block>
R2. <header_list> → <header> <header_list>′
R3. <header_list>′ → <header> <header_list>′
R4. <header_list>′ → ε
R5. <header> → #INCLUDE <HEADER_FILE>
R6. <prog_block> → { <block_stmt> }
R7. <block_stmt> → <dec_list> <stmt_list>
R8. <block_stmt> → <stmt_list>
R9. <dec_list> → <type> <id_list> ;
…

After the grammar rules are built, we build the parse table as shown in table 1.

Table 1. Parsing table

No.	Non Terminals	Input Tokens	Rules	Error Message For Non Terminals And Input Tokens
1	<program>	#	R1	Expected #
2	<header_list>	#	R2	
3	<header_list>′	#	R3	
4	<header_list>′	ε	R4	Skip
5	<header>	#	R5	Expected #
6	<prog_block>	{	R6	Expected {
…	…	…	…	…

5. Implementation and testing

The implementation of this system composed with learning and testing modules as shown in figure 3. For learning Parsing theory and top down parsing algorithm, this system enables user to choose the specific sub menus. For example, here are three sub

menus in Parsing Algorithms Menu. They are Elimination of Left Recursion, First and Follow and Construction of Predictive Parsing Table. User can view the algorithms of the top down parser by choosing the sub menus.

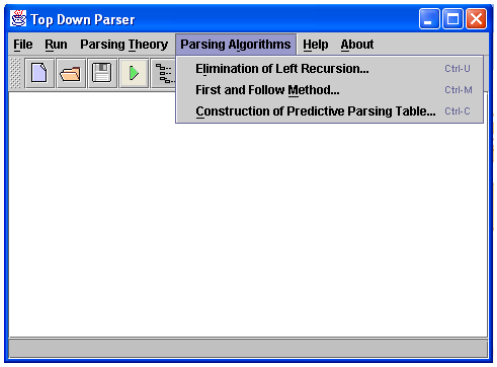

Figure 3. Learn Menu for a non recursive predictive parser

This system allows for parsing the C++ source token stream files. The user can input the new file from the New Menu. For example, token stream file can be seen as follows.

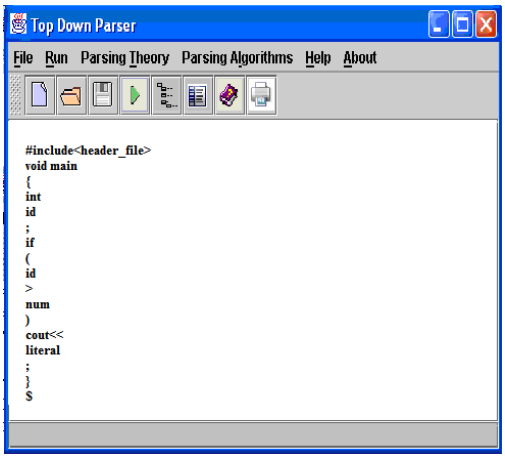

Figure 4. Testing for a non recursive predictive parser

After parsing the correct input token stream file, user can view the syntax tree structure as shown in figure 5. The corresponding grammar rules which are used when top down parser is parsing can be seen as shown in figure 6. The parsed stack information can be seen in figure 7.

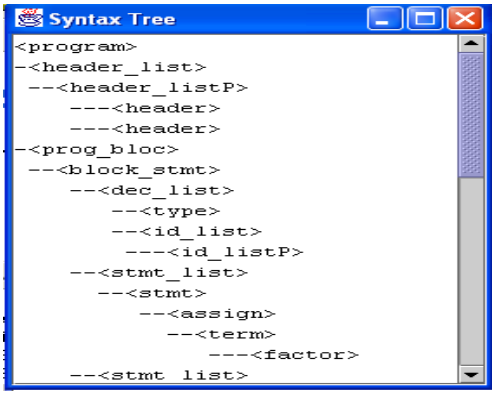

Figure 5. Syntax Tree Structure

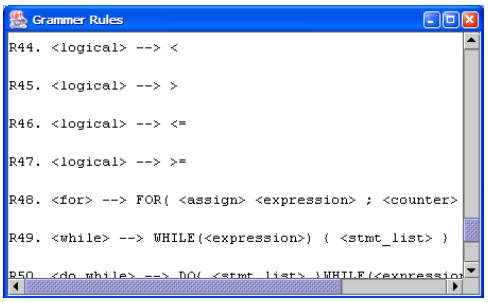

Figure 6. Parsed Grammar Rules

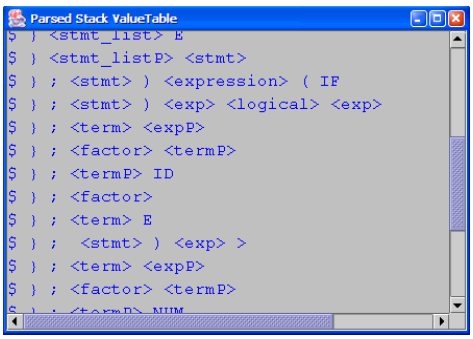

Figure 7. Stack Information

6. Conclusion

This system tries to implement the C++ Top-Down Parser for learning and testing purposes. The input of the Top-Down parser is intended to parse the token streams of the scanner and output is compiled messages and stack information. Especially, it can show the run time used grammar rules and tree structure of the processing steps. And it can also produce the error messages. The popularity of the top down parsers is due to the fact that efficient parsers can be constructed more easily by hand-coded using top-down method.

References

[1] DeRemer, F., “Practical Translators for LR(k) Languages”, Ph.D. dissertation, MIT, 1969.

[2] DeRemer, F., “Simple LR(k) Grammars”, Communications of the ACM, Vol. 14, No. 7, 1971.

[3] Knuth, D., “On the Translation of Languages from Left to Right”, Information and Control, Vol. 8, No. 6, 1965.

[4] Knuth, D., “Top-Down Syntax Analysis”, Acta Informatica, Vol.1, No. 2, 1971.

[5] Korenjak, A., “A Practical Method for Constructing LR(k) Processors”, Communications of the ACM, Vol. 12, No. 11, 1969.

[6] Lewis, P., Rosenkrantz, D., and Stearns, R., “Compiler Design Theory Reading”, MA: Addison-Wesley, 1976.

[7] Lewis, P. and Stearns, R., “Syntax-Directed Transduction”, Journal of the ACM, Vol. 15, No. 3, 1968.

[8] Maggie Johnson and revised by Julie Zelenski, “Miscellaneous Parsing”, CS143 Handout 18 Autumn 2007.

[9] http://en.wikipedia.org/wiki/Semantic_analysis