Implementation Of LL(1) Parser

Aayush Nirwan
1601001

Nilesh Agarwal
1601037

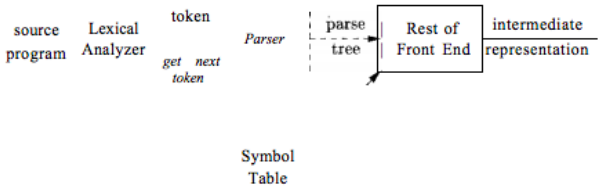
This assingment will give the flavour that how a LL(1) parser works in traditional compilation process.

I. Introduction

The parser obtains a string of tokens from the lexical analyzer, and verifies that if the string of token names can be generated by the grammar for the source language.

There are three general types of parsers for grammars: universal, top-down, and bottom-up. Top-down parsing can be viewed as finding a leftmost derivation for an input string.

The position of the parser in the compiler model can be illustated in the figure below:



II. Context Free Grammar

The grammar which we have implemented is shown below:

(1) MD → type id Med
(2) Med → Brac
(3) Med → Coma ;
(4) Coma → , id Coma
(5) Coma → ϵ
(6) Brac → (){Rep}
(7) Rep → Stmts Rep
(8) Rep → ϵ
(9) Stmts → MD
(10) Stmts → Ret
(11) Stmts → Cond
(12) Stmts → F_loop
(13) Stmts → Job
(14) Job → id Expr ;
(15) Expr → op low Expr
(16) Expr → ϵ
(17) C_E → cond_op Low
(18) Low → lit
(19) Low → id
(20) F_loop → for(idExpr;idC_E;idInr){Stmts}
(21) Inr → ++
(22) Inr → --
(23) Cond → if(Out cond_op Low){Stmts}C'
(24) Out → id Cont
(25) Cont → Expr Cont
(26) Cont → ϵ
(27) C' → ϵ
(28) C' → else {Stmts}
(29) Ret → return Low ;

Note that "MD" is the start symbol. Words starting with capital letter are non-terminal and words starting with small letter are terminals.

III. Implementation

The Implementation of the top-down LL(1) parser can be divided into the following segments:

(a) The productions of the grammar is given in the text file grammar.txt, the terminals and non-terminals are mapped into single characters.

(b) The First and Follow of Non-terminals are given manually in LL1_parser_implementation.c, Using this the LL(1) parsing table is generated by the function generate_parsing_table().

(c) The input file named input.c contains the c code to be parsed, Each token is sent to the previous assignment lexer.h which returns the token type by the implentation of function get_token_type(). It is stored in the array named input_stream.

(d) Stack is initialized by first pushing $ and then Start Symbol, Further the parsing algorithm is carried out.

Each of the Non-Teminals are mapped into upper-case characters and each of the terminals are mapped into lower-case characters.For e.g MD is mapped to A and type is mapped to a. "grammar.txt" also contains the mapping.

reverse_mapping.h -> Contains implementation of get_token_of_symbol(char s) which returns the terminal or non-terminal which is mapped to s.

A. Parsing Table

Algorithm

Algorithm 1 Parsing Table Generator

```
for i <- no_of_productions do
    rule_rhs = rhs_of_production(i)
    for j <- len(rule_rhs) do
        if rule_rhs[j] = epsilon then
req_term = follow(lhs_of_production(i))
        end if
        if rule_rhs[j] = terminal then
If rule_rhs[j] is not present in req_term then add to it
        end if
        if rule_rhs[j] = non_terminal then
req_term = first(lhs_of_production(i))
        end if
    end for
    for k <- len(req_term) do
        table[left_of_production(i)][k] = i
    end for
end for
```

The parsing table is shown in table 1.

B. Parsing

Algorithm

Algorithm 2 Parsing

```
i = 0
flag = 0
input_stream(Stores all the tokens)
push($);push(MD)
while !stack_empty() do
    if stack_top = input_stream[i] then
pop()
i++
    elseif(k=table[stack_top][input_stream[i]])!=1 then
push ( reverse( rhs_of_production(k) ) )
    end if
    else
print "Error"; flag = 1
    end if
end while
if flag == 0 then print "Accepted"
end if
```

IV. Code Analysis

A. Robustness

Robustness describes how reliable our program is, especially under extreme conditions such as extreme workload or unpredictable user inputs. This program would never crash no matter what the user inputs. Even if the user enters invalid data, program will handle it properly. Unpredictable user inputs such as nested for and if loops are even taken care of in the grammar. Even a huge C code will be parsed according to the grammar, generating appropriate errors if any. The program runs even if the number of productions are very high or the grammar is very large (given appropriate first and follow).

B. Correctness

Correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification. Correctness refers to the input-output behaviour of the algorithm. This program might render some of test cases but it will work properly in most of test cases. For example even if there are nested for loops in "input.c" file, the code is parsed. Error handing is taken care properly, for e.g. if the "input.c" file has missing parenthesis or delimiters, the output shows :-
error ——- } is missing.
error ——- ; is missing.
The code is scalable i.e. new grammar can easily be incorporated given correct first and follow.

C. Time Complexity

The time complexity of the LL(1) parser can be calculated as:

Suppose there are n characters in "grammar.txt". *Reading from "grammar.txt" would take :- O(n)*
+
Parsing table generation would take :- O(ab) for table construction + O(pq) for table printing.
a = Number of productions.
b = Max Length of any right hand side of production.
p = Number of Terminals.
q = Number of Non-Terminals.
+
Parsing would take :- O(m)
m = Number of words in "input.c".

D. Space Complexity

The Space Complexity of the LL(1) would be O(n+m).
n = Number of words in input.c
m = Number of characters in grammar.txt(i.e. productions)

V. Error Handling

Errors are handled in a special manner. Along with printing error in the output, program also prints the cause of the error. If the stack top is terminal and an error occurs then that terminal **may** be missing from the code.
Examples of errors detected with a message:-
Parenthesis, Delimeter, Conditional Operator.

Table 1
LL(1) Parsing Table

	type	id	;	,	(	)	{	}	op	cond_op	lit	for	++	–	if	else	return	$
MD	1																	
Med			3	3	2													
Coma			5	4														
Brac					6													
Rep	7	7						8				7			7		7	
Stmts	9	13										12			11		10	
Job		14																
Expr			16						15	16								
C_E										17								
Low		19									18							
F_loop												20						
Inr													21	22				
Cond															23			
Out		24																
Cont									25	26								
C’	27	27						27				27			27	28	27	
Ret																	29	