**Compiler term project – Implementation of a SLR Parser for a simplified C programming language**

Patrik Lytschman (50241262), Lukas Welker (50241179)

Chung Ang University

Compiler (52321)

Hyosu Kim

June 06, 2024

Contents

[Context free grammar (CFG) 3](#_Toc168763353)

[Changes made to the given CFG make it non-ambiguous. 4](#_Toc168763354)

[SLR Parsing Table 6](#_Toc168763355)

[Action 6](#_Toc168763356)

[Goto 7](#_Toc168763357)

[Implementation of the SLR Parser 8](#_Toc168763358)

[Data Structures 8](#_Toc168763359)

[Procedure of the implementation 9](#_Toc168763360)

[Key functions 9](#_Toc168763361)

[Test files and test script 10](#_Toc168763362)

## Context free grammar (CFG)

A Context-Free Grammar (CFG) is a formal system used to describe the syntax of a language. It consists of a set of production rules that define how the symbols of the language can be generated. A CFG is particularly useful in the design and implementation of programming languages and compilers.

A CFG is defined by four components:

1. Non-terminal symbols: These are symbols that can be replaced by other symbols (either terminal or non-terminal) according to the production rules. They represent the syntactic categories in the language
2. Terminal symbols: These are the actual symbols or characters of the language (e.g., letters, digits) that appear in the strings generated by the grammar.
3. Production rules: These are rules that specify how non-terminal symbols can be replaced by sequences of terminal and/or non-terminal symbols.
4. Start symbol: This is a special non-terminal symbol from which the generation of strings begins.

A CFG is typically represented as ,where:

- is a finite set of non-terminal symbols.

- is a finite set of terminal symbols.

- is a finite set of production rules of the form , where is a non-terminal and is a string consisting of terminals and/or non-terminals.

- is the start symbol.

## Changes made to the given CFG make it non-ambiguous.

|  |  |
| --- | --- |
| Given CFG | Non-ambiguous CFG  Also visible in the submitted \*.zip file under files/CFG.txt |
| 1. CODE -> VDECL CODE 2. CODE -> FDECL CODE 3. CODE -> '' 4. VDECL -> vtype id semi 5. VDECL -> vtype ASSIGN semi 6. ASSIGN -> id assign RHS 7. RHS -> EXPR 8. RHS -> literal 9. RHS -> character 10. RHS -> boolstr 11. EXPR -> EXPR addsub EXPR 12. EXPR -> EXPR multidiv EXPR 13. EXPR -> lparen EXPR rparen 14. EXPR -> id 15. EXPR -> num 16. FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace 17. ARG -> vtype id MOREARGS 18. ARG -> '' 19. MOREARGS -> comma vtype id MOREARGS 20. MOREARGS -> '' 21. BLOCK -> STMT BLOCK 22. BLOCK -> '' 23. STMT -> VDECL 24. STMT -> ASSIGN semi 25. STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE 26. STMT -> while lparen COND rparen lbrace BLOCK rbrace 27. COND -> COND comp COND 28. COND -> boolstr 29. ELSE -> else lbrace BLOCK rbrace 30. ELSE -> '' 31. RETURN -> return RHS semi | 1. **CODE -> TYPE CODE** 2. **CODE -> ''** 3. **CODE' -> VDECL** 4. **CODE' -> FDECL** 5. **TYPE -> vtype CODE'** 6. **VDECL -> id VDECL'** 7. **VDECL' -> semi** 8. **VDECL' -> ASSIGN semi** 9. **ASSIGN -> assign RHS** 10. **RHS -> EXPR** 11. **RHS -> literal** 12. **RHS -> character** 13. **RHS -> boolstr** 14. **EXPR -> TERM EXPR'** 15. **EXPR' -> addsub TERM EXPR'** 16. **EXPR' -> ''** 17. **TERM -> FACTOR TERM'** 18. **TERM' -> multdiv FACTOR TERM'** 19. **TERM' -> ''** 20. **FACTOR -> lparen EXPR rparen** 21. **FACTOR -> num** 22. **FACTOR -> id** 23. **FDECL -> id lparen ARG rparen lbrace BLOCK RETURN rbrace** 24. **ARG -> vtype id MOREARGS** 25. **ARG -> ''** 26. **MOREARGS -> comma vtype id MOREARGS** 27. **MOREARGS -> ''** 28. **BLOCK -> STMT BLOCK** 29. **BLOCK -> ''** 30. **STMT -> vtype VDECL** 31. **STMT -> ASSIGN semi** 32. **STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE** 33. **STMT -> while lparen COND rparen lbrace BLOCK rbrace** 34. **COND -> boolstr COND'** 35. **COND' -> comp COND** 36. **COND' -> ''** 37. **ELSE -> else lbrace BLOCK rbrace** 38. **ELSE -> ''** 39. **RETURN -> return RHS semi** |

In order to change our given CFG to a non-ambiguous one, we had to apply various rules and changes. The initial problems was with the derivations 1 and 2 of the given CFG. Here we have a common prefix issue for both derivations. This means we had to eliminate this common prefix. For the elimination of these described ambiguity we had to change the derivation rules 1 - 9, 23 and 30 in our non-ambiguous CFG.

The next problem was with all the derivations of the non-terminal EXPR, this concerned the derivations from our given CFG 11 – 15. In order to make these derivations non-ambiguous we had eliminate left recursion using right recursion and apply left factoring. After applying these steps we got all the derivations rules from in the non-ambiguous CFG 14 – 22.  
 The last ambiguous part of our given CFG concerned all derivation rules of the non-terminal COND. Here we had the problem of left recursion, this concerned the derivations 27 and 28 of our given CFG. After eliminating the left recursion using again right recursion we have changed the derivations rules 34 – 36 in the non-ambiguous CFG.

All other derivations of our given CFG were already non-ambiguous, which means we did not have to change anything at all. Alle the changed derivation rules are marked with green.

## SLR Parsing Table

### Action



### Goto



## Implementation of the SLR Parser

We implemented our SLR parser in python. To execute Python3 must be installed and can be executed by the following command: “python syntax\_analyzer.py <path\_of\_inputfile>”. The input file has to be .txt file containing tokens defined in the Grammar. After the execution either an error message - describing the error - or a parse tree will be printed. The implementation generally follows the idea presented in the script “Syntax Analyzer Part: 5”.

### Data Structures

To represent the SLR parsing table we used dictionaries in python, and we have a dictionary for the actions of the parsing table and a separate dictionary for the goto-statements of the parsing table. The key in these dictionaries is always the current state and therefore each row of the parsing table has its own entry in the dictionary. The value for each key is another dictionary for which the terminals/non-terminals are the key with the corresponding action/goto-statement as value. Additionally, we have a third dictionary in which we store each derivation mapped to its corresponding number in the SLR parsing table. For readability purposes we put the dictionaries in a separate python-file called “ruleset.py” and we import the dictionaries in “syntax\_analyzer.py”.

We also implemented a simple stack in a separate file named “stack.py”. It follows the standard implementation of a stack with the usual methods like pop, peek and push for the stack and a list to store the data.

For the parse tree we also implemented a tree class in the file “tree.py”. Each node has a data attribute to store the data of the node itself and a children attribute which is just a list to store the children’s node. To add a child to the node we have two methods called “add\_child” and “add\_child\_at”. In these the given node will just be stored in the children list at the end or for the specified index. We also added straightforward get methods to obtain the data of the node or the list of children and a “remove\_child” method to remove a child from the children list. Finally, we implemented a method called “print\_tree” which prints first the data of the given node and then recursively the data of the children and we just followed the implementation of a pre-order traversal. We also added additional indents for every recursive call and vertical lines for a nicer looking print in the console.

### Procedure of the implementation

At the beginning of the execution, it will be checked if a path to a file was given or not and if a path was given the tokens in the file are read and stored in a list. Additionally, we store a “$” sign at the end of the list as an end marker for the SLR parser.

After the tokens were successfully obtained and the list of tokens contains at least one token we create two stacks. One is for the states of the SLR parser used later in the parsing process and the second is for the nodes of the tree. Then the function “check\_tokens” is called with the tokens list as an input parameter and if the function verifies the tokens list the root node of the parse tree is returned which will be printed afterwards.

At the beginning of the SLR parsing we set the index pointing at the next input symbol at 0 and set a flag called “accepted” as false. Afterwards follows a while loop that tries to do the SLR parsing as long as the flag is not true, or a syntax error is detected.

In every iteration the current state and the next input symbol is determined by calling the “peek” method of the stack and getting the token from the list at the given index. With the current state and the next input symbol we can get the next action that should be performed from the action dictionary. We then read the first character of the action and if it is “r” we call a reduce function with the action and the index and if it is a “s” we call a shift function with the current action and the index. Also, we check if the action is “acc” and if so, we call a function to accept the input. If there is no action for the given state and next input symbol an error handling function is called.

### Key functions

reduce(): This function is used to perform the reduce operation in the parser. It pops the corresponding number of nodes and states from the stack, creates a new parent node with the non-terminal of the derivation as data and adds the popped nodes as children. Afterwards the new state is determined by the goto dictionary, and the corresponding new state and the parent node are pushed to their respective stacks. We moved the determining of the non-terminal on the left side of the derivation and the number of elements on the right side of the derivation to extra functions to make the code more readable.

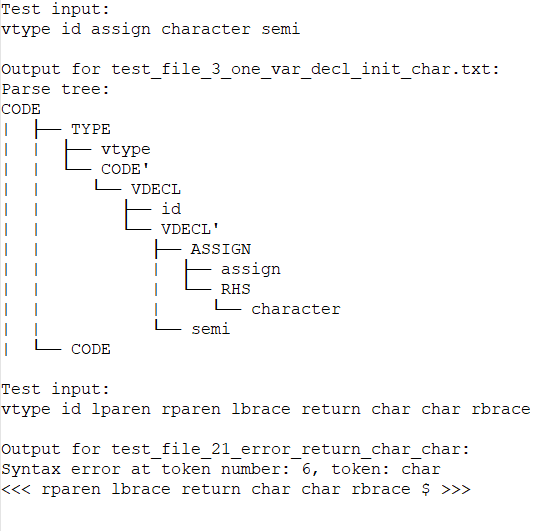
shift(): The shift function just pushes the state given in the action to the stack and creates a new node with the current token as data and pushes it to the node stack. Afterwards we increment the index by one to move the splitter to the right.

error\_handling(): The function is called when a syntax error is determined during the parsing process. In the function we retrieve a maximum of three tokens before and after the index where the error occurred to provide some context for the error. Finally, the function prints out the index at the token where the error occurred and a snippet containing the tokens before and after. Afterwards the program is exited.

accept\_input(): This function is called when the SLR parsing was successful, but we still need to simulate the last derivations in order to obtain the complete parse tree. We simply simulate the derivation “CODE -> TYPE CODE” for the remaining nodes in the stack until only one node remains. We must do this because the SLR parsing table accepts the input without executing the last reductions.

### Test files and test script

To test our SLR parser we created a total of 25 test files based on the specifications of the CFG. The first 19 test files should successfully create parse tree, while the remaining six should detect a syntax error. For an easier testing procedure, we created a script called “test\_script.py” which simply executes the "syntax\_analyzer.py” for all the test files in the directory defined in the config.py file and writes all output into one \*.txt file. Following some examples for our SLR parser:



A screenshot of a computer

Description automatically generated