Practical Assignment - Syntactic Analysis

Ian Nery Bandeira - 17/0144739

University of Brasília - (UnB), P.O. Box 4466, 70910-900
Brasília-DF, Brazil
iannerybandeira@gmail.com

1 Introduction

The Translators course taught at the University of Brasilia focuses on the study of translators and their related elements and implementation. Thus, this subject's primary assignment concerns implementing a translation process for a C language subset combined with complementary methods for handling set theory processes, as the set primitive and operators that cover commonly employed methods for manipulating sets, as well as according to the professor's language description [6]. This paper covers the first three steps of a translator implementation for the previously mentioned C language subset, its lexical, syntactical and semantic analyses.

The patterns that regulate the language grammar, along with the grammar description, are detailed in Appendix A.

2 Description

2.1 Lexical Analysis

To perform the lexical analysis, regular expressions were created to cover the elements described in the language description, such as digits, valid characters for identifiers, keywords, operators (relational, logical, arithmetic, and assignment), comments, and flow control commands. Following this, an analysis was performed regarding the additional tokens that compose the language, such as curly braces, brackets, parentheses, quotation marks, semicolons, and commas.

In order to identify errors with line and column tracking, two variables were also created for this purpose, conducting this inspection line by line. Anything that was not described in the regular expressions defined for the keywords above was considered an "unexpected character" error. The types, identifiers, and delimiter tokens are used to compose the symbol table, and the lexemes are used to create the abstract syntax tree in the syntactic analysis.

As to effectively transfer the line and column tracking variables to the syntactic analysis, along with the token's label, a structure to envelop those parameters was created, and is defined in Listing 1.1.

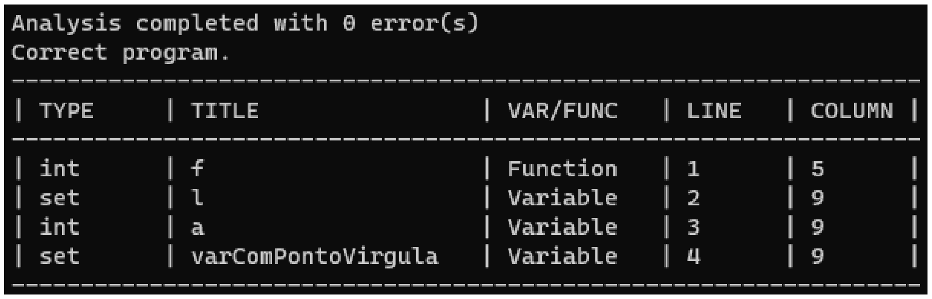
```
struct Token {
      int     t_line;
      int     t_column;
      char    t_title[101];
} token;
```

Listing 1.1. Token Structure.

2.2 Syntactical Analysis

To perform the syntactical analysis, implemented using the Bison parsing tool [3] with a bottom-up parser LR(1) canonical, the patterns and rules for each transition defined in Appendix A as non-terminals, and the tokens were represented as terminals. The main task of this assignment is to define, implement, populate and print a symbol table and an abstract syntax tree.

The definition of a symbol table varies throughout the literature, as to what are the primary or optional fields that are contained within it. The structure of a symbol table is a symbol array, and a symbol is a structure described in listing 1.2. The chosen attributes to be initially displayed in the symbol table are: a symbol's type, their label, whether they are a variable declaration or a function declaration, and their respective declaration line and column. A functional example can be seen in Figure 1.



```
Analysis completed with 0 error(s)
Correct program.
-----------------------------------------------------------------
| TYPE     | TITLE                | VAR/FUNC   | LINE   | COLUMN |
-----------------------------------------------------------------
| int      | f                    | Function   | 1      | 5      |
| set      | l                    | Variable   | 2      | 9      |
| int      | a                    | Variable   | 3      | 9      |
| set      | varComPontoVirgula   | Variable   | 4      | 9      |
-----------------------------------------------------------------
```

Fig. 1. Symbol Table example

```
struct Symbol {
    char    s_type[11];
    char    s_funcvar[11];
    int     s_line;
    int     s_column;
    int     s_scope;
    char    s_title[101];
    int     s_numParams;
    char    s_params[100][31];
} Symbol;
```

Listing 1.2. Symbol Structure.

To implement the abstract syntax tree (AST), it was necessary to implement a dynamic tree with four "children" on each node, as well as a type variable and a token pointer. In order to populate the tree, it was necessary to follow the bottom-up derivation the parser already executes after declaring the grammar, and then to create and link nodes appropriately.

A functional example of both the abstract syntax tree and the symbol table can be executed by typing the following script on a terminal:

```
$ make compile
$ make run
```

2.3 Semantic Analysis

In order to analyze whether the code is semantically correct, it has been considered as endpoints a program that contains/verifies:

- Variables, arguments, and functions scope detection in their declarations and usages;
- Variable typesetting and expression casting when needed;
- Match detection between quantity and typesetting of function arguments and parameters;
- Main function detection.

To implement each endpoint, some previously existing structures were changed, and others were created. As a programmer's choice, I opted to build a two-pass compiler, since some semantic issues were easier to implement during the abstract syntax tree construction, such as the scope detection and the match between function arguments and parameters; while others were easier to build after the AST and the symbol table were complete, such as the main function detection and the expression evaluation/casting in specific cases, in which an expression subtree is traversed in the process of creating the AST.

The biggest change to the already created data structures was to determine the scope validation, in which I used a list that adds and removes items as if it were a stack but can have its elements traversed as a list when necessary to validate each element. It operates so that, when it encounters a '{' symbol, or a function's argument declaration scope, it puts a unique integer in the "stack". Once it encounters the '}' symbol, or finishes declaring arguments of a function, it removes the symbol from the "stack". To analyze whether the variable used was declared in a valid scope, the "stack" is now scrolled through as a list, along with the symbol table, to check if the scope in which the variable was declared matches a scope present in the "'stack".

Other minor changes were made to ensure the typesetting and casting was correct, which added two string attributes to all the node structures in the AST, which contained the type and the corresponding type cast of it's children.

3 Test Files

The test files are stored in the *tests/* folder.

Inside the folder there are 2 files containing correct syntax within the C language subset:

1. t_correct01.c,
2. t_correct02.c;

Along with 2 files containing incorrect syntax:

1. t_error01.c, which contains lexical errors,
2. t_error02.c, which contains syntax errors.
3. t_error03.c, which contains semantic errors.

4 Compilation and Execution Instructions

The syntactic analysis algorithm was compiled and executed with the following system specifications:

- OS Version → Ubuntu 20.04.1 LTS
- Make Version → GNU Make 4.2.1 Built for x86_64-pc-linux-gnu
- Bison Version → bison (GNU Bison) 3.7.6
- Flex Version → flex 2.6.4
- GCC Version → gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1 20.04)

To compile the program, write the following on a terminal:

```
$ make compile
```

To run the executable with the test files, write the following on a terminal:

```
$ ./a.out tests/<test_file_name>
```

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, principles, techniques. Addison wesley **7**(8), 9 (1986)
2. Bagaria, J.: Set Theory. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, spring 2020 edn. (2019), https://plato.stanford.edu/archives/spr2020/entries/set-theory/, visited on 2021-02-16
3. Corbett, R., Stallman, R.: Bison https://www.gnu.org/software/bison/manual/bison.pdf, visited on 2021-03-18
4. Heckendorn, R.: A grammar for the c- programming language (version s21). University of Idaho (2021), http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf, visited on 2021-02-16
5. Kakade, S.: C tokens. Savitribai Phule Pune University (2020), http://studymaterial.unipune.ac.in:8080/jspui/bitstream/123456789/5889/1/C%20Tokens.pdf, visited on 2021-02-16
6. Nalon, C.: Trabalho prático - descrição da linguagem (2021), https://aprender3.unb.br/mod/page/view.php?id=294131, visited on 2021-02-16

A Language Grammar

The patterns, described into Flex Regular Expressions, were created and added to Table 1 according to the valid tokens in the C programming language [5], and the Language Grammar was developed using the C- Grammar [4] as reference, once they are both C language simplified subsets.

1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *varDeclaration* | *funcDeclaration*
4. *varDeclaration* → *simpleVDeclaration* **;**
5. *funcDeclaration* → *simpleFDeclaration* **(** *params* **)** *compoundStmt*
   | *simpleFDeclaration* **( )** *compoundStmt*
6. *params* → *params* **,** *param* | *param*
7. *param* → *simpleVDeclaration*
8. *simpleVDeclaration* → **TYPE ID**
9. *simplefDeclaration* → **TYPE ID**
10. *compoundStmt* → { *stmtList* }
11. *stmtList* → *stmtList primitiveStmt* | *primitiveStmt*
12. *primitiveStmt* → *exprStmt* | *compoundStmt* | *condStmt*
    | *iterStmt* | *returnStmt* | *setStmt* | *inOP* | *outOP* | *varDeclaration*
13. *exprStmt* → *expression* **;**
14. *condStmt* → **if (** *simpleExp* **)** *primitiveStmt*
    | **if (** *expression* **)** *primitiveStmt* **else** *primitiveStmt*
15. *iterStmt* → **for (** *assignExp* **;** *simpleExp*
    ; *assignExp***)** *primitiveStmt*
16. *returnStmt* → **return** *expression* **;**
17. *setStmt* → *forallOP*
18. *pertOP* → *simpleExp* **in ID** | *simpleExp* **in** *setReturner*
19. *setReturner* → *addOP* | *remOP*
20. *typeOP* → **is_set(** *setParams* **)** | **UN_LOGICAL_OP is_set(** *setParams* **)**

21. *setParams* → **ID** | *pertOP* | *setReturner* | *constOP*

22. *addOP* → **add(** *pertOP* **)**

23. *remOP* → **remove(** *pertOP* **)**

24. *selectOP* → **exists(** *pertOP***)**

25. *forallOP* → **forall(** *pertOP* **)** *primitiveStmt*

26. *expression* → *assignExp* | *simpleExp* |*setReturner*

27. *assignExp* → **ID ASSIGN_OP** *expression*

28. *simpleExp* → *binLogicalExp* | *pertOP* | *selectOP* | *typeOP*

29. *constOP* → **INT | FLOAT | EMPTY**

30. *inOP* → **read (ID) ;**

31. *outOP* → **write (outConst) ; | writeln (outConst) ;**

32. *outConst* → **STRING | CHAR |** *simpleExp*

33. *binLogicalExp* → *binLogicalExp* **BIN_LOGICAL_OP** *unLogicalExp*
| *unLogicalExp*

34. *unLogicalExp* → **UN_LOGICAL_OP** *unLogicalExp*
| *relationalExp*

35. *relationalExp* → *relationalExp* **RELATIONAL_OP** *sumExp* | *sumExp*

36. *sumExp* → *sumExp* **SUM_OP** *mulExp* | *mulExp*

37. *mulExp* → *mulExp* **MUL_OP** *factor* | *factor* | **SUM_OP** *factor*

38. *factor* → **ID** | *functionCall* | **(***simpleExp***)** | *constOP*

39. *functionCall* → **ID (***callParams***)** | *functionCall* → **ID ( )**

40. *callParams* → *callParams***,***simpleExp* | *simpleExp*

Table 1. Labels and regular expressions for the language lexemes

Label	Regular Expression(Flex RegEx)
digit	[0-9]
ID	[a-zA-Z][_a-z0-9A-Z]*
EMPTY	EMPTY
KEYWORD	**if|else|for|forall|is_set|return|in|add|remove|exists**
SUM_OP	[+−]
MUL_OP	[∗/]
BIN_LOGICAL_OP	[&]{2}|[|]{2}
UN_LOGICAL_OP	[!]
RELATIONAL_OP	[=]{2}|(! =)|(>=)|(<=)|[>]|[<]
ASSIGN_OP	[=]{1}
INLINE_COMMENT	[/]{2}.∗
TYPE	**int|float|set|elem**
INT	**DIGIT+**
FLOAT	**DIGIT+ . DIGIT+**
STR_DELIM	″
CHAR_DELIM	′