Practical Assignment - Semantic 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 [Nal21]. This paper covers the steps of a translator implementation for the previously mentioned C language subset, with its lexical, syntactical and semantic analyses, and the intermediate code generation.

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.

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. This token structure possesses four attributes:

- Two integers to store the token's line and column;
- A string to store the token's title;
- An integer to store the unique integer generated from the scope handler, which will be further explained in section 2.3.

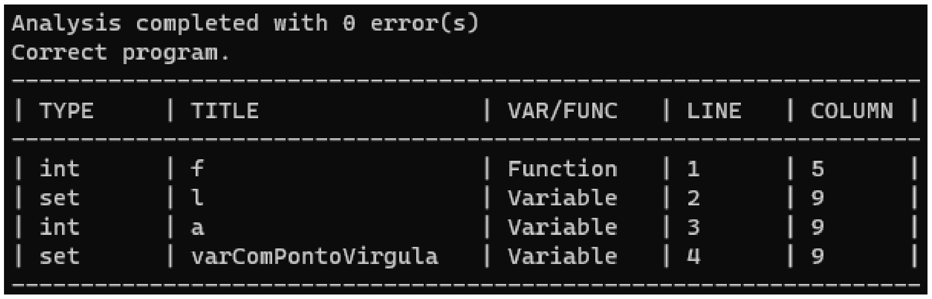
2.2 Syntactical Analysis

To perform the syntactical analysis, implemented using the Bison parsing tool [CS] that generates a bottom-up parser LR(1) canonical, the patterns and rules for each production 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 [ASU86], [Kak02] as to what are the primary or optional fields contained within it. The structure of a symbol table is a symbol array, and a symbol is a structure that contains eight attributes:

- Four integers:
  - Two to store the symbol line/column location inside the file,
  - One to store the integer related to its scope, generated by the scope handler, further explained in section 2.3,
  - One to determine the quantity of function parameters (if the symbol is a function, otherwise is always set to -1);
- Three strings:
  - One to store the symbol's title,
  - One to store its type (between int for Integer, float for Floating Point, elem for Set Element, and set for Set),
  - One to set if the symbol is a function, a variable, or a function parameter;
- And a list of strings to store the function parameters' types (if the symbol is a function, otherwise it is always empty).

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 3.



```
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

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. In contrast, 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 AST creation.

The most significant 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. Whenever a variable or function is declared within this scope, the unique integer at the top of the stack is assigned to its scope attribute. 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.

To ensure the quantity of parameters and arguments of a funcion and a function call, respectively, the parameter quantity and each parameter's type are stored in an array, and then when the function is used, this array is transversed and matched each array element with the parameters. Other minor changes were made to ensure the typesetting and casting were correct, adding two string attributes to all the node structures in the AST, which contained the type and the corresponding typecast of its children. The current typecasting possible is between Integer, Floating pointer, and Element types.

In order to validate the main function existence, the symbol table is transversed to search for a function with the "main" title.

2.4 TAC Generation

The intermediate code generation was based on the TAC (Three Address Code) [San] documentation since the intermediate code could be executed within these guidelines. This intermediate code possesses two main sections:

- **.table**: Used to store the symbol table variables used across the code.
- **.code**: Used to store all functions and expressions used within the code execution.

It is essential to point out that the intermediate code *will not be generated* if there are lexical, syntactic, or semantic errors.

To manage both sections within the already-built AST, two string attributes were added to the node structure, one to store **.table** strings, and other to store **.code** strings. To properly populate both sections, the AST is transversed with a DFS algorithm with postfix commands to write those strings within each section. Some functions are already built inside the code section before the AST transversion, which is necessary to ensure some built-in language methods work (i.e., string write/writeln, add, remove).

To ensure the variables' scope was being displayed correctly in the **.table** section, I concatenated their name along with their unique scope integer. Temporary variables used to execute more complex expressions were assigned to the TAC registers, which vary from $0 through $1023.

The typecasting done to the variables is applied before the intermediate code generation, which concatenates a string that performs the variable/expression typecast inside the intermediate code when needed.

Addressing the most complex operations done and build-in functions added, along with their explanation of how they were implemented:

- **String write/writeln**: This built-in function was implemented through a loop, which printed each string byte (which represents a char), mimicking a string print.
- **Function calls and returns**: For each function declaration, to ensure their names were not any reserved word, I concatenated an underscore with the beginning id of each function. All functions have a return value of zero at the end of their execution, even though it might have other returns inside it, to ensure the function has at least one return. For the *main* function, I also opted to concatenate it with an underscore at its label beginning and returning zero at its end. To solve the TAC error that the main should not have a return, I implemented a *main* label without the concatenated underscore, whose only function is to call the *_main* function and then finish the program.

- **Conditional and Iteration**: Similar to how the functions and temporary variables are used, each if, for, or forall statements were concatenated with an integer to ensure their uniqueness.
- **Set/Elem types**: Elem types were implemented as a three integer vector, in which the first stores the Elem's type, the second store its value, and the last store the next Elem's memory address. The last integer ensures the Set type works since it was implemented as a list of Elems, being that the only value the Set primitive has is the memory address of its first Elem.

3 Test Files

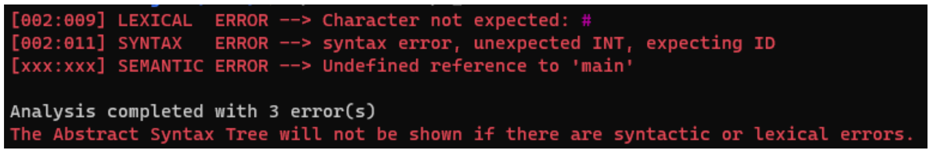
The test files are stored in the *tests/* folder.

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 errors, and their following output:

1. t_error01.c:

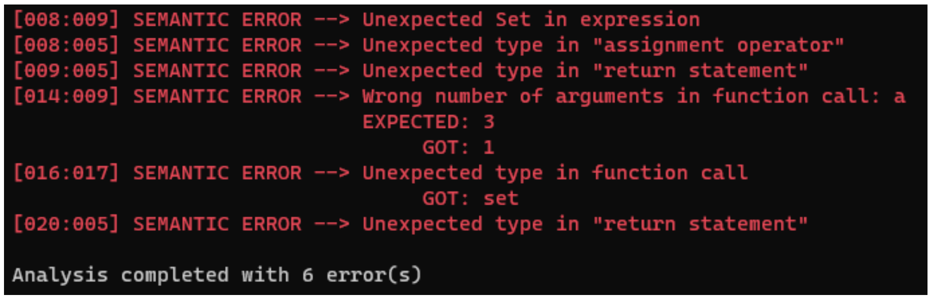


```
[002:009] LEXICAL  ERROR --> Character not expected: #
[002:011] SYNTAX   ERROR --> syntax error, unexpected INT, expecting ID
[xxx:xxx] SEMANTIC ERROR --> Undefined reference to 'main'

Analysis completed with 3 error(s)
The Abstract Syntax Tree will not be shown if there are syntactic or lexical errors.
```

Fig. 2. Error output from t_error01

2. t_error02.c:



```
[008:009] SEMANTIC ERROR --> Unexpected Set in expression
[008:005] SEMANTIC ERROR --> Unexpected type in "assignment operator"
[009:005] SEMANTIC ERROR --> Unexpected type in "return statement"
[014:009] SEMANTIC ERROR --> Wrong number of arguments in function call: a
                             EXPECTED: 3
                                  GOT: 1
[016:017] SEMANTIC ERROR --> Unexpected type in function call
                                  GOT: set
[020:005] SEMANTIC ERROR --> Unexpected type in "return statement"

Analysis completed with 6 error(s)
```

Fig. 3. Error output from t_error02

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 generate the TAC output file, run the executable with the test files with the following command on a terminal:

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

To execute the TAC output file, install the TAC executable through `github.com/lhsantos/tac`, and execute it with the output file:

```
$ ./tac out.tac
```

References

ASU86. Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

Bag19. Joan Bagaria. Set Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2020 edition, 2019. visited on 2021-02-16.

CS. Robert Corbett and Richard Stallman. Bison. visited on 2021-03-18.

Hec21. Robert Heckendorn. A grammar for the c- programming language (version s21). *University of Idaho*, 2021. visited on 2021-02-16.

Kak02. O. G. Kakde. *Algorithms for Compiler Design*. Charles River Media, Inc., USA, 1 edition, 2002.

Kak20. Swapnali Kakade. C tokens. *Savitribai Phule Pune University*, 2020. visited on 2021-02-16.

Nal21. Cláudia Nalon. Trabalho prático - descrição da linguagem, 2021. visited on 2021-02-16.

San. Luciano Santos. Simple three address code virtual machine. `https://github.com/lhsantos%7D%7D`. visited on 2021-03-18.

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 [Kak20], and the Language Grammar was developed using the C- Grammar [Hec21] as reference, once they are both C language simplified subsets.

1. $program \rightarrow declarationList$
2. $declarationList \rightarrow declarationList\ declaration \mid declaration$
3. $declaration \rightarrow varDeclaration \mid funcDeclaration$
4. $varDeclaration \rightarrow simpleVDeclaration$ **;**
5. $funcDeclaration \rightarrow simpleFDeclaration$ **(** $params$ **)** $compoundStmt$
   $\mid simpleFDeclaration$ **( )** $compoundStmt$
6. $params \rightarrow params$ **,** $param \mid param$
7. $param \rightarrow simpleVDeclaration$
8. $simpleVDeclaration \rightarrow$ **TYPE ID**
9. $simplefDeclaration \rightarrow$ **TYPE ID**
10. $compoundStmt \rightarrow$ **{** $stmtList$ **}**
11. $stmtList \rightarrow stmtList\ primitiveStmt \mid primitiveStmt$
12. $primitiveStmt \rightarrow exprStmt \mid compoundStmt \mid condStmt$
    $\mid iterStmt \mid returnStmt \mid setStmt \mid inOP \mid outOP \mid varDeclaration$
13. $exprStmt \rightarrow expression$ **;**
14. $condStmt \rightarrow$ **if** **(** $simpleExp$ **)** $primitiveStmt$
    $\mid$ **if** **(** $expression$ **)** $primitiveStmt$ **else** $primitiveStmt$
15. $iterStmt \rightarrow$ **for** **(** $iterAssign$ **;** $iterExp$ **;** $iterAssign$**)** $primitiveStmt$
16. $iterAssign \rightarrow assignExp \mid \varepsilon$
17. $iterExp \rightarrow simpleExp \mid \varepsilon$
18. $returnStmt \rightarrow$ **return** $expression$ **;**
19. $setStmt \rightarrow forallOP$
20. $pertOP \rightarrow simpleExp$ **in** $factor$

21. *setReturner* → *addOP* | *remOP*

22. *typeOP* → **is_set(** *setParams* **)**

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

24. *addOP* → **add(** *pertOP* **)**

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

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

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

28. *expression* → *assignExp* | *simpleExp*

29. *assignExp* → **ID ASSIGN_OP** *expression*

30. *simpleExp* → *binLogicalExp*| *pertOP*

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

32. *inOP* → **read (ID) ;**

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

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

35. *binLogicalExp* → *binLogicalExp* **BIN_LOGICAL_OP** *unLogicalExp* | *unLogicalExp*

36. *unLogicalExp* → **UN_LOGICAL_OP** *unLogicalExp* | *relationalExp*

37. *relationalExp* → *relationalExp* **RELATIONAL_OP** *sumExp* | *sumExp*

38. *sumExp* → *sumExp* **SUM_OP** *mulExp* | *mulExp*

39. *mulExp* → *mulExp* **MUL_OP** *signedFactor* | *signedFactor*

40. *signedFactor* → *factor* | **SUM_OP** *factor*

41. *factor* → **ID** | *functionCall* | **(***simpleExp***)** | *constOP* | *selectOP* | *typeOP* | *setReturner*

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

43. *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	‘