Syntax Analyzer*

Marcella Pantarotto - 13/0143880

University of Brasília (UnB), Brasília DF, Brazil
marcellapantarotto@gmail.com

1 Motivation

The study on operation and construction of compilers is fundamental for computer scientists, because they can understand how high-level languages are converted into machine code to be executed. This understanding allows for techniques that can be adopted in code development to produce good performance and memory-saving programs.

The language selected for this project is called C-IPL and is very similar to the C language [Nal21]. It differs because it has new primitives to assist in the treatment of lists. There is a focus on the study of lists because they are a fundamental data structure for software development, with very important functionalities. Among the new primitives added to the language there are two types of constructors and some operators used for the manipulation of the list data or to return specific values from the list. Without these primitives, list implementation and manipulation becomes more difficult because the handling of lists in the C language uses simpler data structures along with dynamic memory allocation.

2 Description of the Lexical Analyzer

The developed lexical analyzer is created based on the language C-IPL, [Nal21], and with the help of Flex, a tool for creating scanners that parses and recognizes lexical patterns in text [PEM16]. For that, several regular expressions were declarated to compose the rules of a scanner. Each one of these rules helps the lexical analyzer to parse the input code and identify all tokens. Tokens which are considered the smallest valid part of a program are composed by two parts: its name and its attributes. The attributes can be information about the type of the token, its position inside the symbol table, its scope, among other things. They are displayed in the following format: <**TOKEN, ATTRIBUTE**>.

Some tokens are detected by the keywords displayed in the table of the Appendix C and other by some regular expressions, better detailed in the table of Appendix B, but briefly explained bellow:
- **number:** digits from 0 to 9
- **letter:** A to Z (in upper and lower case)
- **white_space:** to recognize separation of tokens

* Thanks to Professor Cláudia Nalon

- **int:** integer numbers
- **float:** floating point numbers
- **string:** segment of text declared between double quotes
- **comment/comments:** segments that can be ignored by the parser
- **id:** composed of letters, numbers and underscore, to recognize identifiers

The identified keywords are: **main**, **return**, **int**, **float**, **list**, **if**, **else**, **for**, **write**, **writeln**, **read**. The new primitives are: ":", "?", "!", "%", "≫", "≪". Other expressions were also used to detect and handle error cases. During parse, every time the lexical analyzer finds an identifier, which can be a variable or a function name, it stores it into the symbol table. This table is a fundamental part of a compiler [ALSU03]. Tokens not recognized by the scanner are considered errors, because they are expressions that were not declared in the language [Lev09].

Two structs were used to construct the symbol table, one storing the individual information for each symbol and another storing a pointer to the beginning and another pointer to the end of the table. Using this information a storage structure with a linked list was built. Each symbol inside the table has an index, a title and a pointer to the next symbol. When initializing the table, the start and end pointers point to the same element and as symbols are added to the table, the end pointer changes its direction to the newly added symbol.

3 Description of the Syntax Analyzer

The syntax analyzer was built with the help of Bison [DS15], a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR (left-right) parser. In this context, a canonical LR(1) parser table was created to interpret the tokens recognized by the lexical analyzer.

Both parsers are integrated, the tokens recognized by the lexical analyzer are passed to the syntactic analyzer which, from the grammar described in Appendix A, assembles a tree structure to check the correctness of the input program.

Three structs are used to construct the tree. The first is to store the information of the tokens that are passed by the lexical analyzer. The second is to store the information of each node in the tree and the third is an auxiliary, which stores a pointer to a node and another pointer to the sibling nodes that are on the same level, which are accessed through a linked list.

The information stored for each node is: its token, which can be empty if it is a rule node, its type, and a pointer to its list of child nodes, also accessed with a linked list. There are two types of nodes in the syntactic tree, one called "node" that represents the derivative rules of the grammar and another type called "token" that represents the terminals derived from these rules.

An important function of a compiler is the scope detection within a program, this is done by the scanner and parser together and with the help of an auxiliary tree structure. Whenever the parser finds opening curly brackets, it adds a node to the scope tree and if more opening curly brackets are found, subsequent child

nodes are added under the first node found. When the scanner finds closing curly brackets, it goes back a node level inside the scope tree. When a variable or a function is declared, the lexical analyzer adds it to the symbol table, along with the scope information. This process will help the semantic analysis, if it finds a symbol it looks in the symbol table for the scope information to ensure that no violations are being made, such as a variable being redeclared within the same scope.

4 Description of Test Files

There are four test files and they are all located in the "tests" directory inside the project. The tests **test_correct1.c** and **test_correct2.c** have no syntactical errors. The test **test_errors1.c** has errors regarding an extra type *int* (line 2, column 9) and a variable declaration with a following attribution (line 3, column 11). The test **test_errors2.c** has errors regarding a variable declaration with a following attribution (line 2, column 22) and a declaration of a list pop operation with two identifiers (line 4, column 5). These tests with errors help in the analysis of how the syntax analyzer works.

5 Compilation and Execution Instructions

A Makefile was created to help with compilation and execution of the project, so the first command is for compilation and the second is for execution.

```
$ make
$ make run
```

To change the file executed, inside the Makefile, <**input_file**> should be replaced by the input file.

```
$ ./tradutor tests/<input_file>
```

References

[ALSU03] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: principles, techniques and tools, 2nd ed.* Pearson Education India, 2003.

[DS15] Charles Donnely and Richard Stallman. Gnu bison: the yacc-compatible parser generator. https://www.gnu.org/software/bison/manual/, 2015. (Last acessed on 02 Sep 2021).

[Lev09] John Levine. *Flex & Bison: Text Processing Tools, 1st ed.* "O'Reilly Media, Inc.", 2009.

[Nal21] Cláudia Nalon. Description of C-IPL language for college project (2021). https://aprender3.unb.br/mod/page/view.php?id=464034, Aug 2021. (Last acessed on 09 Aug 2021).

[PEM16] Vern Paxson, Will Estes, and John Millaway. The flex manual, for flex 2.6.2. https://westes.github.io/flex/manual/, Oct 2016. (Last acessed on 09 Aug 2021).

Appendices

A Grammar of the C-IPL Language

⟨*program*⟩ ::= ⟨*list_declarations*⟩
| ε

⟨*list_declarations*⟩ ::= ⟨*declaration*⟩ ⟨*list_declarations*⟩
| ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*function_declaration*⟩
| ⟨*variable_declaration*⟩

⟨*function_declaration*⟩ ::= ⟨*unique_declaration*⟩ **(** ⟨*parameters*⟩ **)** **{**⟨*block_commands*⟩**}**

⟨*variable_declaration*⟩ ::= ⟨*unique_declaration*⟩ **;**

⟨*unique_declaration*⟩ ::= ⟨*type*⟩ **identifier**

⟨*parameters*⟩ ::= ⟨*list_parameters*⟩
| ε

⟨*list_parameters*⟩ ::= ⟨*unique_declaration*⟩ **,** ⟨*list_parameters*⟩
| ⟨*unique_declaration*⟩

⟨*calling_parameters*⟩ ::= ⟨*list_calling_parameters*⟩
| ε

⟨*list_calling_parameters*⟩ ::= ⟨*operation*⟩ **,** ⟨*list_calling_parameters*⟩
| ⟨*operation*⟩

⟨*block_commands*⟩ ::= ⟨*command*⟩ ⟨*block_commands*⟩
| ⟨*command*⟩

⟨*command*⟩ ::= ⟨*variable_declaration*⟩
| ⟨*init_variable*⟩
| ⟨*conditional_stmt*⟩
| ⟨*return_stmt*⟩
| ⟨*iteration_process*⟩
| ⟨*input_operation*⟩
| ⟨*output_operation*⟩
| **{** ⟨*block_commands*⟩ **}**
| ⟨*operation*⟩ **;**

⟨*init_variable*⟩ ::= ⟨*init_stmt*⟩ **;**

⟨*conditional_stmt*⟩ ::= **if (** ⟨*operation*⟩ **)** ⟨*command*⟩
| **if (**⟨*operation*⟩**)** ⟨*command*⟩ **else** ⟨*command*⟩

⟨*return_stmt*⟩ ::= **return** ⟨*operation*⟩ **;**

⟨*iteration_process*⟩ ::= **for (**⟨*loop_condition*⟩**)** ⟨*command*⟩

⟨*loop_condition*⟩ ::= ⟨*init_stmt*⟩ **;** ⟨*operation*⟩ **;** ⟨*update_stmt*⟩

⟨*init_stmt*⟩ ::= **identifier** = ⟨*operation*⟩

⟨*update_stmt*⟩ ::= ⟨*init_stmt*⟩
| ε

⟨*input_operation*⟩ ::= **read (** ⟨*expression*⟩ **) ;**

⟨*output_operation*⟩ ::= **write (** ⟨*operation*⟩ **) ;**
| **writeln (** ⟨*operation*⟩ **) ;**
| **write (string) ;**
| **writeln (string) ;**

⟨*function_calling*⟩ ::= **identifier (** ⟨*calling_parameters*⟩ **) ;**

⟨*expression*⟩ ::= ⟨*function_calling*⟩
| ⟨*single_operation*⟩
| ⟨*constant*⟩
| **identifier**

⟨*constant*⟩ ::= ⟨*number*⟩
| **NIL**

⟨*number*⟩ ::= **number_int**
| **number_float**

⟨*type*⟩ ::= ⟨*type_list*⟩
| ⟨*type_number*⟩

⟨*type_list*⟩ := ⟨*type_number*⟩ **list**

⟨*type_number*⟩ ::= **int**
| **float**

⟨*operation*⟩ ::= ⟨*arithmetic_binary*⟩
| ⟨*list_binary*⟩
| ⟨*operation*⟩ ⟨*relation_operator*⟩ ⟨*expression*⟩
| ⟨*operation*⟩ ⟨*logic_operator*⟩ ⟨*expression*⟩

⟨*single_operation*⟩ ::= ⟨*arithmetic_single*⟩
| ⟨*list_single*⟩
| ! ⟨*expression*⟩

⟨*arithmetic_binary*⟩ ::= ⟨*arithmetic_binary*⟩ + ⟨*expression*⟩
| ⟨*arithmetic_binary*⟩ − ⟨*expression*⟩
| ⟨*arithmetic_binary*⟩ ∗ ⟨*expression*⟩
| ⟨*arithmetic_binary*⟩ / ⟨*expression*⟩
| ⟨*expression*⟩

⟨*arithmetic_single*⟩ ::= + ⟨*expression*⟩
| − ⟨*expression*⟩

⟨*list_binary*⟩ ::= ⟨*expression*⟩ : ⟨*expression*⟩
| ⟨*expression*⟩ ≫ ⟨*expression*⟩
| ⟨*expression*⟩ ≪ ⟨*expression*⟩

⟨*list_single*⟩ ::= ? ⟨*expression*⟩
| % ⟨*expression*⟩

⟨*logic_operator*⟩ ::= **&&**
| ||

⟨*relational_operator*⟩ ::= >
| ≥
| <
| ≤
| ==
| ! =

B Lexical Grammar of Regular Expressions

Some terminals of the grammar are expressed in their own format. The keywords are expressed with their exact name, as an example, the *return* expression has the lexeme "return". There are also symbols that are terminals and represent operations, assignment, delimitation of scope and punctuation. Some of these terminals were shown in Section 2 but they are specified in the table in Appendix C. Bellow there is a table that specifies the terminals that are expressed with regular expressions.

Token	**Regular Expression (Regex)**	**Pattern**
number	[0-9]	digits
letter	[a-zA-Z]	letters (lowercase and uppercase)
white_space	[\t\v\f\r]+	all types of white spaces
int	{number}+	integer constant numbers
float	{number}+[.] | (({*number*} ∗ [.])?{*number*}+)	floating point constant numbers
string	\"(\\. | [^"\\])*\"	strings (sequence in double quotes)
comment	\/\/,*	inline comment
comments	\/*(,*\n)*.**\/	multi-line comment
id	[_a-zA-Z][_a-zA-Z0-9]*	identifiers

C Lexical Grammar of Keywords and Other Expressions

Token	Lexeme	Pattern
NIL	“NIL”	keyword (constant for empty list construction)
main	“main”	keyword (main function)
return	“return”	keyword (return statement)
int	“int”	keyword (type int)
float	“float”	keyword (type float)
list	“list”	keyword (type list)
if	“if”	keyword (conditional statement)
else	“else”	keyword (conditional statement)
for	“for”	keyword (iteration)
write	“write”	keyword (output)
writeln	“writeln”	keyword (output and jump line at the end)
read	“read”	keyword (input)
addition	“+”	arithmetic operation (addition)
subtraction	“-”	arithmetic operation (subtraction)
multiplication	“*”	arithmetic operation (multiplication)
division	“/”	arithmetic operation (division)
equal	“==”	relational operation (equal)
less than	“<”	relational operation (less than)
grater than	“>”	relational operation (grater than)
less or equal than	”<=”	relational operation (less or equal than)
grater or equal than	“>=”	relational operation (grater or equal than)
different than	“!=”	relational operation (different than)
and	“&&”	logical operation (and)
or	“||”	logical operation (or)
assignment	“=”	assignment statement
ambiguous statement	“!”	logical operation (not) OR list operation (tail)
header	“?”	list operation (header)
constructor	“:”	list operation (construct list)
destructor	“%”	list operation (removing first element and returning tail)
map	“>>”	list operation (map)
filter	“<<”	list operation (filter
punctuation symbols	“(” | “)” | “{” | “}” | “,” | “;”	parentheses, curly brackets, comma and semicolon
break line	“\n”	breaking from one line to another