Lexical 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. And 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 2 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 Analyser

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 declared 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 are: **<TOKEN, ATTRIBUTE>**. The regular expressions used were:
- number: digits from 0 to 9
- letter: A to Z (in upper and lower case)
- white_space: to recognize separation of tokens
- 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

[*] Thanks to Professor Cláudia Nalon

The keyword identified are: **main**, **return**, **int**, **float**, **list**, **if**, **else**, **for**, **write**, **writeln**, **read**. The new primitives are: ":", "?", "!", "%", "$\gg$", "$\ll$".

Other expressions were also used to detect and handle error cases. During parse, every time the lexical analyzer finds an identifier, it stores it into the symbol table. This table was created using linked lists and is a fundamental part of a compiler [AUS03]. Tokens not recognized by the scanner are considered errors, because they are expressions that were not declared in the language [Lev09]. The output displayed on the console identifies the row and column of each element, being either a valid token or an error. It also shows the symbol table created to store all the identifiers and the total number of lexical errors, both found in the input file.

3 Description of Test Files

There are 4 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 lexical errors. The test **test_errors1.c** has errors regarding an invalid extra point (line 2, column 13) and an invalid identifier declaration (line 3, column 9). The test **test_errors2.c** has errors regarding an invalid expression (line 6, column 27) and an invalid point (line 8, column 6). These tests with errors help in the analysis of how the lexical analyzer works.

4 Compilation and Execution Instructions

Inside the project's main directory there is a **readme.txt** file that explains in more detail all the necessary steps for compiling and running the project, as well as instructions on how to run each one of the tests. But for a brief overview, the first two commands are needed for compilation and the third is needed for test execution. <**test_file_name**> should be replaced by the test files mentioned in the previous section.

```
$ flex -o src/lex.yy.c src/lex.l
$ gcc -g -Wall src/lex.yy.c -o tradutor
$ ./tradutor tests/<test_file_name>
```

References

[AUS03] Alfred Aho, Jeffrey Ullman, and Ravi Sethi. *Compilers: principles, techniques and tools, 2nd ed.* Pearson Education India, 2003.

[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 of declarations*⟩

⟨*list of declarations*⟩ ::= ⟨*declaration*⟩⟨*list of declarations*⟩
| ⟨*declaration*⟩
| ⟨ε⟩

⟨*declarations*⟩ ::= ⟨*variable declaration*⟩
| ⟨*function declaration*⟩

⟨*variable declaration*⟩ ::= ⟨*type*⟩ ⟨*ID*⟩**;**

⟨*function declaration*⟩ ::= ⟨*type*⟩ ⟨*ID*⟩ **(**⟨*parameters*⟩**)** ⟨*function definition*⟩

⟨*function declaration*⟩ ::= **{**⟨*block of commands*⟩ **return** ⟨*value*⟩**;}**

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

⟨*parameter*⟩ ::= ⟨*type*⟩ ⟨*ID*⟩

⟨*block of commands*⟩ ::= ⟨*variable declaration*⟩ ⟨*block of commands*⟩
| ⟨*command*⟩ ⟨*block of commands*⟩
| ⟨ε⟩

⟨*command*⟩ ::= ⟨*assignment command*⟩
| ⟨*conditional command*⟩
| ⟨*iteration command*⟩
| ⟨*function calling*⟩
| ⟨*return command*⟩

⟨*assignment command*⟩ ::= ⟨*ID*⟩ **=** ⟨*value*⟩

⟨*conditional command*⟩ ::= **if(**⟨*condition*⟩**)** **{**⟨*block of commands*⟩**}**
| **if(**⟨*condition*⟩**)** **{**⟨*block of commands*⟩**}** **else** **{**⟨*block of commands*⟩**}**

⟨*iteration command*⟩ ::= **for** **(**⟨*condition*⟩**)** **{**⟨*block of commands*⟩**}**

⟨*function calling*⟩ ::= ⟨*ID*⟩**(**⟨*parameters*⟩**);**

⟨*return command*⟩ ::= **return** ⟨*value*⟩**;**

⟨*type*⟩ ::= **int**
| **float**
| **list**

⟨*value*⟩ ::= ⟨*ID*⟩
| ⟨*constant*⟩
| ⟨*string*⟩

⟨*constant*⟩ ::== ⟨*number*⟩
| ⟨*letter*⟩
| **NIL**

⟨*logic-operators*⟩ ::= **!**
| **&&**
| ||

⟨*relational-operators*⟩ ::= $>$
| $\geq$
| $<$
| $\leq$
| $==$

⟨*arithmetic-operators*⟩ ::= $+$
| $-$
| $*$
| $/$

⟨*list-operators*⟩ ::= :
| ?
| !
| %
| $\gg$
| $\ll$

⟨*token*⟩ :: = ⟨*token, attribute*⟩

⟨*int*⟩ ::= ⟨*number*⟩+

⟨*float*⟩ ::= ⟨*number*⟩+**.**
| **.**⟨*number*⟩+
| ⟨*number*⟩***.**⟨*number*⟩+

⟨*id*⟩ ::== [**_a-zA-Z**][**_a-zA-Z0-9**]*****

⟨*letter*⟩ ::= [**a-zA-Z**]

⟨*number*⟩ ::= [**0-9**]

B Lexical Grammar of 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	\"(\\\. | [ˆ"\\\])*\"	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 (input)
writeln	“writeln”	keyword (input and jump line at the end)
read	“read”	keyword (output)
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
punctiation symbols	“(” | “)” | “{” | “}” | “,” | “;”	parentheses, curly brackets, comma and semicolon
break line	“\n”	breaking from one line to another