Lexical Analyzer*

Marcella Pantarotto[13/0143880]

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

Abstract. First practical assignment for the Translators course at the University of Brasília. This is the first part of a larger project that will compose a compiler. As the first essential part of a compiler, this project dealt with the construction of the lexical analyzer, which identifies the tokens of a code written in the C-IPL language.

Keywords: Compiler · Lexical Analyzer · Tokens.

1 Motivation

The translators course is offered at the University of Brasilia in order to expand the knowledge about compilers. In this course the functioning of compilers is studied, as well as the best practices for building one. This knowledge is fundamental for computer scientists, as they can understand how high-level languages are converted into machine code to be executed. And this understanding allows for the elaboration of techniques that can be adopted in code development to produce good performance and memory-saving programs.

2 Description of the Lexical Analyser

The developed lexical analyzer is created based on the language C-IPL, described in [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 expression 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, considered the smallest unit of a program. The regular expressions used were:
- **number:** digits from 0 to 9
- **letter:** A to Z (in upper and lower case)
- **white-space:** to recognize separtion 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
- **id-error:** to recognize invalid types of identifiers

* Thanks to Professor Cláudia Nalon

Other expressions were also used to detect keywords and handle other error cases, as well as new primatives: “?”, “%”, “≪” , “≫”, “:”, “!”. 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 tha were not declared in the language [Lev09]. These errors are presented in the console, along with their respective row and column. The output of the parser also shows all the tokens identified, also with their row and column. The output also displays a view of the symbol table created to store all the identifiers of the input code.

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), an invalid identifier declaration (line 3, column 9), and an invalid expression (line 8, column 12). The test **test_errors2.c** has errors regarding an invalid extra point (line 4, column 15), an invalid identifier declaration, and an invalid expression (line 6, column 37) 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).

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

⟨*assingment 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*⟩**;**

⟨*types*⟩ ::= '**int**'|'**float**'|'**list**'

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

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

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

⟨*relational-operators*⟩ ::= >
| ≥
| <
| ≤
| ==

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

⟨*int*⟩ ::= ⟨*number*⟩+

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

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

⟨*letter*⟩ ::= **[a-zA-Z]**

⟨*number*⟩ ::= **[0-9]**