Lexical Analysis

Igor Figueira Pinheiro da Silva - 15/0129921

University of Brasília

1 Motivation and Language Description

The work presented here was developed for the Compilers course from the University of Brasília. The work will be divided into four steps: lexical analysis, syntactic analysis, semantic analysis, and intermediate code generation. Our target language is called C-IPL, which is a simplified version of the C programming language with a newly introduced *list* type. The following resources are introduced:

- **Types:** *float list* and *int list*
- **? operator:** used for accessing the list head
- **! operator:** used for accessing the list tail without modifying the list
- **% operator:** used for accessing the list tail and removing the first element of the list
- **>> operator:** infix binary operator which receives an unary function as first argument and a list as the second argument. It returns a new list after **mapping** the input list using the input function.
- **<< operator:** infix binary operator which receives an unary function as first argument and a list as the second argument. It returns a new list after **filtering** the input list using the input function.

2 Lexical Analysis

Lexical analysis is the first phase of the compilation process [ALSU06]. At the start we receive the source code as input. This source code is processed, patterns are recognized by using regular expressions which will tell if the lexemes that are being processed belong to the C-IPL language or not and then output a stream of tokens. For this first phase we are not returning the tokens. Instead, we print them in the console following the <*token name, token value*> format.

The lexical analyzer used was generated using Flex [Est]. The tokens and informal description of the regular expressions used in the lexical analysis can be found at table 1.

The symbol table was not implemented yet but we plan to use a hashmap data structure as a base design for it since the records should be found as fast as possible. In our flex source file a function called **update_position** was added to compute the current column and current line in order to give the user a feedback when lexical errors are found.

3 Testing

The files for testing the lexical analyzer can be found attached in the **tests** folder. Source files correct1.c and correct2.c are supposed to work without any errors. File wrong1.c has the following errors:

- Token not recognized: "@". Line: 7, Column: 6
- Token not recognized: "#". Line: 8, Column: 6
- /* never ending comment block at line 10

There is also a second incorrect file called wrong2.c, which has the following errors:

- Token not recognized: "$". Line: 5, Column: 10
- Token not recognized: "". Line: 7, Column: 12
- Token not recognized: "". Line: 7, Column: 42
- Error: " at line 8, column 11 does not have a closing "
- Error: " at line 9, column 11 does not have a closing "
- Error: " at line 10, column 11 does not have a closing "

4 Compiling and Executing

A Makefile is provided inside the main folder. To run it just use the **make** command in your terminal. In case you can't use make you can use the following commands from the **15_0129921** directory:

```
$ flex src/lex.l
$ gcc *.c -Wall -o tradutor
```

In order to execute any of the example files run:

```
$ ./tradutor < tests/chosen_example.c
```

References

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[Est] Will Estes. Lexical Analysis With Flex, for Flex 2.6.2. `https://westes.github.io/flex/manual/`. [Online; accessed 19-August-2021].

[Pol] Bary W Pollack. BNF Grammar for C-Minus. `http://www.csci-snc.com/ExamplesX/C-Syntax.pdf`. [Online; accessed 19-August-2021].

Attachment 1 - Context Free Grammar

1. program → declaration-list
2. declaration-list → declaration-list declaration | declaration
3. declaration → var-declaration | func-declaration
4. var-declaration → data-type **ID SEMICOLON**
5. data-type → **INT_TYPE**
   | **FLOAT_TYPE**
   | **INT_LIST_TYPE**
   | **FLOAT_LIST_TYPE**
6. func-declaration → data-type **ID LPARENTHESES** params-list **RPARENTHESES** block-statement
7. params-list → params | ε
8. params → params **COMMA** param | param
9. param → data-type **ID**
10. block-statement → **LBRACE** statement-or-declatarion-list **RBRACE**
11. statement-or-declaration-list → statement-or-declaration-list statement
    | statement-or-declaration-list var-declaration
    | ε
12. statement → expression-statement
    | block-statement
    | conditional-statement
    | iteration-statement
    | return-statement
    | input-statement
    | output-statement
13. expression-statement → expression **SEMICOLON** | **SEMICOLON**
14. conditional-statement → **IF_KW LPARANTHESES** expression **RPARANTHESES** statement | **IF_KW LPARENTHESES** expression **RPARENTHESES** statement **ELSE_KW** statement
15. iteration-statement → **FOR_KW LPARENTHESES** expression **SEMICOLON** expression **SEMICOLON** expression **RPARENTHESES** statement
16. return-statement → **RETURN_KW SEMICOLON** | **RETURN_KW** expression **SEMICOLON**
17. input-statement → **READ_KW LPARENTHESES ID RPARENTHESES**
18. output-statement → write-call **LPARENTHESES** simple-expression **RPARENTHESES COMMA**
19. write-call → **WRITE_KW** | **WRITELN_KW**
20. expression → **ID ASSIGNMENT** expression | simple-expression
21. simple-expression → math-expression relational-operator math-expression
    | math-expression binary-logical-operator math-expression
    | **NOT_OR_TAIL_OP** math-expression
    | math-expression
    | list-expression

22. relational-operator → **LESSTHAN_OP**
    | **LESSEQUAL_OP**
    | **GREATERTHAN_OP**
    | **GREATEREQUAL_OP**
    | **NOTEQUAL_OP**
    | **EQUAL_OP**
23. binary-logical-operator → **AND_OP** | **OR_OP**
24. list-expression → **LIST_HEAD_OP** math-expression
    | **LIST_TAIL_OP** math-expression
25. math-expression → math-expression add-sub-operator term | term
26. add-sub-operator → **ADD_OP** | **SUB_OP**
27. term → term mul-div-operator factor | factor
28. mul-div-operator → **MULT_OP** | **DIV_OP**
29. factor → **LPARENTHESES** expression **RPARENTHESES**
    | func-call
    | **ID**
    | **INT_CONST**
    | **FLOAT_CONST**
    | **LIST_CONST**
30. func-call → **ID LPARENTHESES** args-list **RPARENTHESES**
31. args-list → args | ε
32. args → args **COMMA** expression | expression

Attachment 2 - Lexical rules for obtaining tokens

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
INT_TYPE	int	int
FLOAT_TYPE	float	float
INT_LIST_TYPE	int list	int list
FLOAT_LIST_TYPE	float list	float list
INT_CONST	+,- followed by integer	-1, 10, 45
FLOAT_CONST	+,- followed by floating point number	-0.1, .5, 45.67
LIST_CONST	NIL	NIL
STRING_CONST	characters inside double quotes	”string”
ADD_OP	+	+
SUB_OP	-	-
MULT_OP	*	*
DIV_OP	/	/
NOT_OR_TAIL_OP	!	!
OR_OP	||	||
AND_OP	&&	&&
LIST_HEAD_OP	?	?
LIST_TAIL_OP	%	%
LIST_CONSTRUCTOR_OP	:	:
LIST_MAP_OP	>>	>>
LIST_FILTER_OP	<<	<<
LESSTHAN_OP	<	<
LESSEQUAL_OP	<=	<=
GREATERTHAN_OP	>	>
GREATEREQUAl_OP	>=	>=
NOTEQUAL_OP	!=	!=
EQUAL_OP	==	==
LBRACE	{	{
RBRACE	}	}
LPARENTHESES	(	(
RPARENTHESES	)	)
SEMICOLON	;	;
ASSIGNMENT	=	=
COMMA	,	,
FOR_KW	for	for
IF_KW	if	if
ELSE_KW	else	else
RETURN_KW	return	return
READ_KW	read	read
WRITE_KW	write	write
WRITELN_KW	writeln	writeln
ID	letter([a-zA-Z]) or underscore(_) followed by letters, digits([0-9]) and underscores	a, b, _variable, two_names

Table 1. Tokens used by the lexical analyzer