Syntax 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 list data structure is used in a large number of applications. By introducing our new primitive we facilitate the developer's implementation by introducing lists as a primitive type of our language. We also provide operators that are list specific and that can be used by the list type. The newly introduced resources are:

- **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 a 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 a 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.

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.

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 scope in the C-IPL language can be verified in the lexical analysis. Every time an opening brace (**LBRACE** token) is found a new scope starts. And every time a closing brace (**RBRACE** token) is found, we go back to the previous scope. This is needed when building our symbol table, which is described in 4.1 Symbol Table.

3 Syntax Analysis

Syntax Analysis, also known as parsing, is the second phase of the compiler [ALSU06]. The parser uses a context free grammar to check if the the string of tokens produced by the lexical can be generated and. When the grammar cannot generate a string using the given tokens, the parser reports a syntax error. If the grammar is able to generate all given strings then we can say the parsed code has a correct syntax.

During the parsing phase we build an abstract syntax tree. Nodes can be terminal symbols representing attribution or an additive operation, for example, and leaves are the operands, such as an identifier or a numeric constant. Details of the implementation can be see at 4.2 Abstract Syntax Tree.

The parser used in our compiler was generated using Bison [CT]. The definition of the grammar used by our parser can be seen at Attachment 1 - Context Free Grammar.

4 Data Structures

This section describes implementation details for our symbol table and abstract syntax tree data structures.

4.1 Symbol Table

The symbol table is used by compilers to hold information about source program constructs [ALSU06]. We have a column to store the identifier, another column that stores a variable type or the keyword function in case its a function identifier and lastly, we have the scope information. As it was mentioned in Lexical Analysis the scope of each symbol table entry is defined during the lexical analysis. We define a new scope for each block, which is defined by opening and closing braces.

Instead of having a single table with a column defining the scope, we opted to have a symbol table for each scope. Each symbol table also points to the inner scope symbol tables in a tree-like data structure. By doing this, we can search for a symbol table entry in a inner scope and if we can't find the entry, we just need to search for it in the parent node recursively. If the entry is not found even at global scope we identify a missing variable declaration for such identifier.

The identifiers for each table entry are registered during the syntax analysis. In rule 4 we are able too get variable identifier values. In rule 6 we can get function identifier values. In rule 5 we can get the data type or return type for such identifiers.

4.2 Abstract Syntax Tree

The abstract syntax tree or syntax tree is an abstract representation of the input. Unlike the parse tree, the abstract syntax tree captures the syntax structure of the input in a much simpler way [Slo]. As of right now the tree is only being built by the parser, but later on it will be used in the semantics analysis as well.

The syntax tree is built solely by the parser by making a good use of its bottom up parsing. When we find grammar rules that only have a single terminal symbol, a leaf is built containing information about that symbol. If we have a grammar rule which has more than one symbol or has mixed terminal and nonterminal symbols a node and is chained to other nodes or leaves according to the grammar rule.

The following piece of code defines our node definition. We store a name that represents the node. This could be either an identifier, an operand or an intermediate grammar rule name. Next and prev are defined to make this data structure list iterable by using the utlist library [Han]. And finally we have node_list which is list of pointers to the children of that node.

```
// Node definition for AST
struct node {
  char* name;
  node_t* prev;
  node_t* next;
  node_t* node_list;
};
```

5 Testing

The files for testing the syntax 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:

- syntax error, unexpected LPARENTHESES, expecting LBRACE at line 4 col 12
- syntax error, unexpected IF_KW at line 6 col 5

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

- syntax error, unexpected ASSIGNMENT, expecting LPARENTHESES or SEMICOLON at line 1 col 7
- syntax error, unexpected LPARENTHESES at line 4 col 10
- Token not recognized: "&". Line: 7, Column: 6

6 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 cannot use make you can use the following commands from the **15_0129921** directory:

```
$ bison -d -o src/syn.tab.c src/*.y; \
  flex --outfile=src/lex.yy.c src/*.l; \
  gcc src/*.c -Wall -Ilib/  -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.

CT. R. Corbett and GNU Project Team. Bison manual. https://www.gnu.org/software/bison/manual/. [Online; accessed 1-September-2021].

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

Han. T. D. Hanson. utlist: linked list macros for C structures. https://troydhanson.github.io/uthash/utlist.html. [Online; accessed 1-September-2021].

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

Slo. K. Slonneger. Syntax and Semantics of Programming Languages. https://homepage.divms.uiowa.edu/~slonnegr/plf/Book/. [Online; accessed 1-September-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 SEMICOLON**
18. output-statement → write-call **LPARENTHESES** output-arg **RPARENTHESES SEMICOLON**
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-constructor
    | list-func
    | **LIST_TAIL_OP ID**
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
    | numeric-const
    | **LIST_HEAD_OP ID**
    | **ID**
    | **LIST_CONST**
30. func-call → **ID LPARENTHESES** args-list **RPARENTHESES**
31. args-list → args | ε
32. args → args **COMMA** expression | expression
33. list-constructor → list-constructor-expression **LIST_CONSTRUCTOR_OP ID**
34. list-constructor-expression → list-constructor-expression **LIST_CONSTRUCTOR_OP** math-expression | math-expression
35. list-func → list-func-expression list-func-operator **ID**
36. list-func-expression → list-func-expression list-func-operator **ID** | **ID**
37. list-func-operator → **LIST_MAP_OP** | **LIST_FILTER_OP**
38. numeric-const → add-sub-operator **FLOAT_CONST**
    | add-sub-operator **INT_CONST**
    | **FLOAT_CONST**
    | **INT_CONST**
39. output-arg → simple-expression | **STRING_CONST**

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	unsigned integer	10, 45
FLOAT_CONST	unsigned floating point number	.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