Translator[⋆]

Marcella Pantarotto - 13/0143880

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

1 Motivation

The study on operation and construction of compilers and translators 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 in the Appendix C and other by some regular expressions, better detailed in the table in 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 be classified as identifiers

The identified keywords are: **return**, **int**, **float**, **list**, **if**, **else**, **for**, **write**, **writeln** and **read**. The new primitives for lists are: ":" for construction, "?" to return the head element of the list, "!" to return list's tail, "%" to remove the first element of the list, "$\gg$" to apply a map function to the list, "$\ll$" to filter an element of the list and also the constant **NIL** which is a keyword to express that a list is empty.

Other expressions were also used to detect and handle error cases. During scanning, 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 lexical 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 initially an index, the name of the symbol 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 sequence of tokens recognized by the lexical analyzer.

Both scanner and parser 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 structure, 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 children 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 the Semantic Analyzer

So far we have seen the steps of lexical analysis, which breaks the source program into tokens, and syntactic analysis, which validates the rules and syntax of the programming language. However, it is not possible to represent some rules with regular expressions or with a context-free grammar. So, because of this, a semantic analysis is performed validating a number of rules that cannot be checked in the previous steps, in order to ensure that the source program is coherent and can be converted to machine language. This is the third stage of the compilation process and it is responsible for checking aspects related to the meaning of the instructions.

Many validations must be performed with meta information and with elements that are present at various points in the source code, far away from each other. To correlate the meta information, the semantic analyzer uses the syntactic tree and the symbol table to perform the semantic analysis. No new structures were built, only functions to perform the verifications of the semantic analysis rules, these rules are explained ahead.

So while the syntactic tree is being built the semantic analyzer collects some information from some nodes of the tree and stores them in the symbol table. The information stored is the type of each symbol, if it is **int**, **float**, **list (int)** or **list (float)**, differentiate which symbols were variables, functions or variables being passed as parameters and also the amount of parameters required in each function. The semantic analyzer also checks if the amount of parameters passed in the function call is equal to the amount of parameters expressed in the function declaration. For that the semantic analyzer calculates how many parameters there are in the function declaration and writes this value to the symbol table. Then, when a function is called, it calculates how many parameters are being passed, and checks the symbol table to see if the number matches the required number. If it doesn't, an error is thrown with a message pointing whether the amount passed is higher or lower than expected.

While the syntactic tree is being assembled, the semantic analyzer keeps track whether to throw an error if there is an attempt to redeclare a variable or function within the same scope. It also checks the scope in which a variable is being used: if it is being used in a place out of their scope, then an error is thrown. The semantic analyzer also checks to see if the variable used or the function called has in fact been declared, to do this it goes through the symbol table to see if it exists and if not, an error is thrown.

An important component of semantic analysis is type checking, in which the compiler checks whether each operator receives the operands allowed and specified in the source language. An example that illustrates this type validation step very well is the assignment of objects of different type. In some cases, the compiler automatically converts from one type to another that is suitable for the operator's application. In this project the semantic analyzer recognizes if the type passed in the return of a function is the same type expected by the function, if it is not, an error is thrown. Also, it converts integer number into floating point numbers during arithmetic operations and relational operations. During logical operations, the result is considered to be an integer.

Finally, the semantic analyzer also goes through the symbol table to verify the existence of a main function in the program being analyzed, because it is a mandatory requirement of the C-IPL language. All these rules and verifications are extremely important for the next step of the compiler, the generation of intermediate code, which is performed in two steps. The first step is the construction of the symbol table and abstract tree and the error checking to ensure that the code is ready for compilation, this step was detailed in Sections 2, 3 and 4 and the symbol table and the annotated abstract tree are displayed in the output shown in the terminal. The second step is to go through the tree created in the previous step and convert the code from C-IPL into a machine-independent typed language and send it to and output file, so it can be interpreted by the TAC tool [SN15], this is better detailed in the next section.

5 Description of the Intermediate Code Generator

The last step of the compilation process is the generation of the intermediate code, which happens in two steps. During the previous steps the information necessary for the generation of the intermediate code was added in the symbol table and also in the abstract tree. To reach this stage, the input file must be free of lexical, syntactic and semantic errors. After the project is executed by passing an input file "*.c*", an output file "*.tac*" is generated and serves as input for the TAC tool [SN15] that executes the code.

The TAC tool is a Three Address Code interpreter, which is a simple turing-complete utility for loading and interpreting a language specification based on the high-level code [ALSU03], in simpler words, it interprets the intermediate code [SN15].

This code is composed of two sections. An example of how the pattern of *tac* syntax is displayed bellow:

```
.table
<type> <name>
<type> <name>[ ] = { <initializer> }
<type> <name>[<vector_size>] = { <initializer> }

.code
<op>: <param>
<op>: <param>, <param>
<op>: <param>, <param>, <param>
[label] <op>: <param>
[label] <op>: <param>, <param>
[label] <op>: <param>, <param>, <param>
```

Inside the *.table* section, all the symbols from the symbol table are added with their type and this happens during the first step. In the *.code* section are all the translated instructions that manipulate the data and execute the code, this part is added in the second step.

To write the second part (*.code* section), the program, during the second parse, adds the commands to an auxiliary stack structure, which is used to store all the instructions from the input program. Then the instructions are removed from the stack directly to the output *.tac* file. This method was adopted to ensure that the correct order of the instructions was preserved.

For the new list primitives, dynamic memory allocation is used through the *tac* commands: ***mema*** and ***memf***. In the ***mema*** command a value is passed and this value will be the size of the list. This size is dynamically allocated and the symbols are located contiguously to simulate a list. The ***memf*** command deallocates these contiguously allocated spaces. Therefore, it helps in handling the data for the new primitives.

6 Description of the 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 or semantical errors. The test **test_errors1.c** has syntactical errors regarding an extra type *int* (line 2, column 9) and a variable declaration with a following attribution (line 3, column 11). And regarding semantical errors, this test has no declaration of function **main** and a variable is being redeclared inside the same scope (line 10, column 11).

The test **test_errors2.c** has syntactical errors regarding a declaration of a list pop operation with two identifiers (line 17, column 10) and a unexpected **else** statement, without **if** statement (line 18, column 5). And regarding semantical errors, this test has a total of three errors, two about wrong passage of parameters in a function call, in the first the amount of parameters being passed is lower then expected (line 19, column 15) and in the second the amount is higher then expected (line 22, column 17). There is also an error regarding a wrong type passing in the return statement of the function main (line: 23, column: 12).

These tests with errors help in the analysis of how the syntax and the semantic analyzers work.

7 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>
```

The commands to compile and execute without the Makefile are:

```
$ gcc-11 -o obj/structures.o -g -c src/structures.c
$ bison --defines=lib/syn.tab.h src/syn.y -o src/syn.tab.c
$ flex -o src/lex.yy.c src/lex.l
$ gcc-11 -g -Wall -o tradutor -I lib/ src/syn.tab.c src/lex.yy.c
$ ./tradutor tests/<input_file>
```

After that, a "*.tac*" file is generated in the *tests* directory with the same name as the input file. To run it you must have the TAC tool executable, one is available inside the project main directory. But if it doesn't work in your machine, you should enter the TAC github directory [SN15], download the project and run it to generate the executable. Once that is done, get the executable and put it in the main directory of the project. The output "*.tac*" of the project, that was created inside the *tests* directory, will run as an input in the TAC executable by running the following command:

```
   $ ./tac tests/<tradutor_output>.tac
```

The project was developed under the following environment (for instructions on how to install or update any item, please have a look at the file"readme.txt" inside the main project directory):

1. Operational system: WSL Ubuntu version 20.04
2. Compiler: GCC version 11
3. Flex version 2.6.4
4. Bison version 3.7

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).

[SN15] Luciano Santos and Cláudia Nalon. TAC - three address code interpreter. https://github.com/lhsantos/tac, Mar 2015. (Last acessed on 24 Oct 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*⟩**}**

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

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

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

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

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

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

⟨*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*⟩
 | ⟨*output_operation*⟩
 | ⟨*input_operation*⟩

| **{** ⟨*block_commands*⟩ **}**
| ⟨*operation*⟩ **;**

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

⟨*init_stmt*⟩ ::= ⟨*iden*⟩ = ⟨*operation*⟩

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

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

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

⟨*loop_condition*⟩ ::= ⟨*update_stmt*⟩ **;** ⟨*stop_stmt*⟩ **;** ⟨*update_stmt*⟩

⟨*update_stmt*⟩ ::= ⟨*init_stmt*⟩
| ε

⟨*stop_stmt*⟩ ::= ⟨*operation*⟩
| ε

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

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

⟨*iden*⟩ ::= **identifier**

⟨*operation*⟩ ::= ⟨*list_binary*⟩
| ⟨*operation*⟩ $>$ ⟨*operation*⟩
| ⟨*operation*⟩ $\geq$ ⟨*operation*⟩
| ⟨*operation*⟩ $<$ ⟨*operation*⟩
| ⟨*operation*⟩ $\leq$ ⟨*operation*⟩
| ⟨*operation*⟩ $==$ ⟨*operation*⟩
| ⟨*operation*⟩ $! =$ ⟨*operation*⟩
| ⟨*operation*⟩ **&&** ⟨*operation*⟩
| ⟨*operation*⟩ $||$ ⟨*operation*⟩

⟨*list_binary*⟩ ::= ⟨*list_binary*⟩ $\gg$ ⟨*list_binary*⟩
| ⟨*list_binary*⟩ $\ll$ ⟨*list_binary*⟩
| ⟨*list_binary*⟩ : ⟨*list_binary*⟩
| ⟨*arithmetic_binary*⟩

⟨*arithmetic_binary*⟩ ::= ⟨*arithmetic_binary*⟩ $+$ ⟨*term*⟩
| ⟨*arithmetic_binary*⟩ $-$ ⟨*term*⟩
| ⟨*term*⟩

⟨*term*⟩ ::= ⟨*term*⟩ $*$ ⟨*expression*⟩
| ⟨*term*⟩ / ⟨*expression*⟩
| ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*function_calling*⟩
  | ⟨*single_operation*⟩
  | ⟨*constant*⟩
  | ⟨*iden*⟩
  | **(** ⟨*operation*⟩ **)**

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

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

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

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

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

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	\”(\\\. | [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)
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