c7: C for set's operations

Dayanne Fernandes da Cunha

Department of Computer Science, University of Brasilia
130107191@aluno.unb.br

Abstract. This language aims to enable set operations using C language syntax with an easy and natural notation.

Keywords: C · Language Processor · Compiler · Set.

1 Proposal

Sets are a very common data structure in many fields, such as Mathematics, Biology, Chemistry, and Computer Science. It is a collection of elements that follow a certain property, and with them, it is possible to describe many complex and simple problems. For example, a set in Mathematics can be used to describe geometrical shapes and algebraic components. In Computer Science it is used to perform logical operations that are fundamental to the computer's existence itself.

This language aims to extend the C language to support set operations with the news data types, **set** and **elem**, using C's syntax and inheriting its semantics rules. Also, it will support simple arithmetic operations, read and write commands, functions, and flow structures. For further information about what syntax the compiler supports please read Appendix B. The book [1] will be used as a guide for the implementation of c7's compiler architecture.

2 Lexical Analyzer

2.1 Architecture

The lexical analyzer is called "**lexer**". This module receives a character stream, analysis it trying to find *lexemes* related to patterns and constructs tokens to send to the parser module. The input needs to match the *c7* formal definitions defined in Appendix A, e.g. the input "int var = 1 + 1" will generates the following output <int> <id, 'var'> <assign, '='> <integer, '1'> <add, '+'> <integer, '1'>. The pair <**token,** ***lexeme***> represents the information that will be sent to the parser.

2.2 Error Handler

Lexemes that are not recognized from any regular expression of the language will be shown in the compiler output as a *LexerError*, showing the line and column indexes that this character/pattern was not correctly identified, e.g.:

```
columns |1|2|3|4|5|6|7|8|9|10|
line 1. |v|a|r|_|2| |+|=| |@|
```

```
Line 1: <id, 'var_2'> <add, '+'> <assign, '='>
LexError: token '@' is not recognized in line 1, column 10.
```

The system does not exit immediately after a lexical error is found, instead, it recovers and searches for other errors in the source code until the end of the characters stream.

2.3 Symbol Table

The library *uthash* [3] is used to create the symbol table structure. Each symbol has a structure *word* that contains a *key* of type integer, a char array called *name* with a limit of 50 characters, an integer called *id_type* to flag the id as function or variable, and an instance of an internal object from *uthash* *UT_hash_handle*, called *hh*.

There are five functions to help add, delete, edit and find words in the symbol table, void add_word(int key, char *name), struct word *find_word(int word_key), void set_id_type(int key, int id_type), void delete_word(struct word *s) and void delete_all_st(), in addition there are two helpers functions to show the symbol table and to count its elements, void print_st() and int len_st() respectively. During the lexer process the symbol table will only keep the names of variables and functions.

2.4 Code Structure and Custom Functions

FLEX [5] 2.6.4 was used to build the lexical analyzer. Some of the internal Flex functions were used to read a character stream from a file and define the tokens and patterns. This *Flex* definition of *c7* is defined in the file **lexer/c7.lex**. All the important files concerning the lexer analyzer are located inside of the folder **lexer**.

The system starts with the function *int main (int argc, char* argv*[ ]*)* inside of the file **core/main.c** and reads a file as input. Every pattern recognized goes through a pipeline inside of the analyzer defined as follows: "{*<PATTERN>*} { *handle_token(<PATTERN_TOK>); return <PATTERN>*};". The function *void handle_token(int token)* is a switch that is responsible to handle the valid tokens and send to the parser, increment the lines and columns as the source code is read, and in addition, detects and shows any lexical error that is found in the code.

3 Syntax Analyzer

3.1 Architecture

The syntax analyzer in *c7* is called "**parser**". It is a module that receives two inputs, the tokens recognized and the symbol table initialized by the lexer. As output, the parser will update the symbol table with more relevant information about the identifiers, e.g., type and scope, and the parser also generates the abstract syntax tree. There are three types of parsers that can be used to match the string patterns, universal, top-down and bottom-up. This proposal will use the **bottom-up** approach **Canonical LR** from Bison.

3.2 Error Handler

When there is a syntax error, the parser emits the token found and the ones expected by the defined grammar, also the line and column that this syntax error appeared in the source file. Example:

```
SyntaxError: syntax error, unexpected ADD_SET, expecting
                               SEMICOLON in line 5, column 8.
```

The system does not exit immediately after the first syntax error is found, instead, just like the lexer, it recovers and searches for other errors in the source code until the end of the characters stream.

3.3 Symbol Table and Abstract Syntax Tree

During the parser process, a new attribute is set in the symbol table's words, this attribute is called *id_type*. There are only two possible types, *ST_ID_FUNC*, and *ST_ID_VAR*, that are used to flag an identifier as a function or variable respectively.

The AST (Abstract Syntax Tree) is generated by the parser using the grammar rules (see syntax grammar in Appendix B). The AST structure is implemented as a linked list of structs that contains a integer variable called tag to flag the struct with the type of the node's expression and an union to represent non terminals and terminals.

3.4 Code Structure

GCC, version 9.3.0 and Bison [2] 3.7.5 were used to build the parser analyzer. The *Bison* definition of *c7* is defined in the file **parser/c7.y**. There are many custom functions to help manage the *AST* located at the file **core/ast.c**. For example, there is a type of functions that follows the pattern *"ast_node* create_[TYPE]_expr ()"*, where [*TYPE*] stands for the node expression type, created to build a node in the *AST*. It can be of the type: *int*, *float*, *var*, *char*, *str*, *binary*, *ternary*, *quartenary*, *quinary*, *function* and *type cast*.

4 Semantic Analyzer

4.1 Architecture

In this step of the translation, it will be analyzed a few semantic rules in one pass, which means that at the same time the lexer and parser are processing the rules will be checked. Information regarding the scope of the variables, flow control, and functions are added in the symbol table in this module. Also, if an implicit type conversion is found, a new node is added above the number/ID's node with the following description, “float2int” or “int2float” in the *AST*.

4.2 Semantic Rules

There are six semantic rules defined in the *c7* language:

- The source code should contain only one **main** function;
- It is not possible to declared a variable or a function more than once;
- A variable and a function that is not declared should not be used or called respectively;
- The parameters of a function call needs to match the arity of the function's declaration;
- A variable cannot be used as a function, e.g. “int x; x();”;
- Implicit type cast occurs between float and integer, other rules of type casting are not supported.

4.3 Error Handler

When a semantic rule is not respected, the translator raises an error during the lexer/parser process, so the messages are shown in the screen before the symbol table and *AST* appear. There are six types of errors that can be raised by the semantic analyser, they are:

- SemanticError: ‘main’ function was not found in the source code;
- SemanticError:[line]:[column]: [symbol] does not match the function declaration. The function call contains [value] parameters and ‘[symbol]’ was declared with [value] parameters;
- SemanticError:[line]:[column]: ‘[symbol]’ was used as a function but ‘[symbol]’ was declared as a variable in line [line], column [column];
- SemanticError:[line]:[column]: ‘[symbol]’ was not declared;
- SemanticError:[line]:[column]: ‘[symbol]’ was already declared in line [line] column [column]. This symbol belongs to the scope ‘[scope]’, lvl [value];
- SemanticError:[line]:[column]: Expression with wrong implicit type cast, type [type].

4.4 Code Structure

GCC, version 9.3.0 was used to build the semantic analyzer. The code used to create the scope and analyze the semantic rules were defined in the file **core/scope.c** and **core/scope.h**. Some custom functions helped managing the symbol table and *AST* during the semantic process that were added to the files **core/ast.c**, **core/ast.h**, **core/sym_tab.c** and **core/sym_tab.h**. The library *utstack* [4] was added to manage the scope dynamic stack structure. The most important procedures will be described in the following subsection.

4.5 Scope and Symbol Table

In this process of the translator, it is important to know if a symbol was declared before is used, if it is not being declared again, if there is a main function in the source code, etc. To handle all these questions a dynamic stack structure was made to support scope managing. There are two main structures to control the scope, a global symbol table and a stack of symbol tables.

When the source code starts to be processed, when a function is declared, and when a bracket is open, a new scope structure is created. A new scope contains a level number, the scope name, a symbol table, and a pointer to the next scope in the stack.

4.6 Type Casting and AST

The *AST* is annotated with a middle node between an expression and a number or a variable indicating an implicit type casting, only from integer to float or float to integer. In the case of expressions assignments, the operands will be converted to the variable data type, e.g. "int x; x = 1.0", the operand 1.0 will be converted to an integer, a node **float2int** will be added in the *AST*. In this step of the translation, only the type cast nodes are added, the conversion will be implemented later.

The function's return is also checked to see if there is an implicit conversion to make, the expression type from the syntax rule ***RETURN*** *expression* is compared with the function type, when the type is an integer or float the conversion is implicit, otherwise, a semantic error will be raised. A semantic error is also raised in case of arithmetic operations that combines integer or float with set or elem.

5 Intermediate Code Generator

5.1 Architecture

This is the last step of the translator, where the *AST* and *symbol table* are ready to be transformed into an intermediate code. This module generates a file with the format *tac* that can be executed by the tool *TAC* [6]. This file is only generated in case there is no error raised in the previous steps. Each type of expression's node built in the Section 3 is translated into the *tac* syntax that follows the pattern below:

```
.table
<type> <name> ['['<vector_size>']'] ['='<initializer>]\n
.code
[label] <name> [<param>] [','<param>] [','<param>]\n
```

5.2 Code Structure

GCC, version 9.3.0 and *TAC* [6], last commit 8e84250c34558e71d9ec6b255c15 71d7deddb6e9, was used to build the intermediate code generator. The code used to create and modify the file generated were defined in the file **tac/builder.c** and **tac/builder.h**. The important functions in these files are "*void write_table(...)*", "*void write_node_instruction(...)*", "*void write_code(...)*" and "*void close_tac(...)*". Some customs functions built to manage the *AST*, located in **core/ast.c** and **core/ast.h**, were modified to generate the code along the lexer/parser process, such as "*ast_node* create_bin_expr(...)*" where some routines were added to build the node's expression intermediate code.

5.3 AST, .table and .code

Two new important parameters were added in the *AST* structure, "char *c ode_instruc" and "int code_register", also a global variable "int global_register". The first parameter is used to store the tac's code generated depending on the node's type and the second is a register that represents that node respectively. The global variable "int global_register" is a counter of registers used in the translator.

The code is generated in one pass during the lexer and parser process, but just at the end of it, the file is created and the instructions in the table section are add looping through the global symbol table. The instructions in the code section are add using a postorder tree traversal algorithm in the *AST*.

6 Usage Manual

To compile, run and test memory leaks in the c7 language follow the instructions below. It is preferable to use a Linux OS to run this translator. These tests were executed using *Flex* 2.6.4, *Bison* 3.7.5, *GCC* 9.3.0, *Make* 4.2.1, *TAC* [6], last commit 8e84250c34558e71d9ec6b255c1571d7deddb6e9, and *Valgrind* 3.15.0.

```
# Compile, run and test a valid example
cd src
make clean
make
./c7 tests/general/valid_1.c7
tac runner.tac
```

```
# Test memory leaks
make clean
make
export TEST_FILE=tests/general/valid_1.c7
make valgrind
```

7 Tests

After executing the usage steps shown in Section 6, there are four files to test the lexical, syntactic, and semantic cases. The files "src/tests/general/valid_1.c7" and "src/tests/general/valid_2.c7" showcase valid patterns in all three aspects and generates the file "src/runner.tac". The files "src/tests/general/invalid_1.c7" and "src/tests/general/invalid_2.c7" show invalid characters, structures and semantics's cases.

There are eight errors in the file "invalid_1.c7", they are:

- Line/column 1:5, lexical error, long symbol;
- Line/column 1:57, syntactic error, due to the first error, the expression "int [long_id];" is not complete as valid structure;
- Line/column 10:9, semantic error, func_2 call before declaration;
- Line/column 14:13, semantic error, func_3 does not match the function declaration;
- Line/column 18:14, lexical error, & is not a valid lexeme;
- Line/column 18:17 and 18:18, syntactic error, due to & not being recognized as valid lexeme, the expression "test & f" is not complete as valid structure inside of the if expression;
- Line/column 32:20, the return type is a set and the function data type is an integer;
- There is no main.

There are four errors in the file "invalid_2.c7", they are:

- Line/column 2:12, syntactic error, it is not possible to define and declare a variable in the same expression;
- Line/column 4:9, lexical error, @ is not a valid lexeme;
- Line/column 5:9, semantic error, because of the first error x was not declared;
- Line/column 8:13, semantic error, z is a variable and it was used as a function.

In case of any lexical and/or syntactic error, the *AST* will not be printed on the screen, and in case of any error at all the file "runner.tac" is not generated.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
2. Corbett, R.: Bison 3.7.1, `https://www.gnu.org/software/bison/manual/bison.html`, last accessed on 31/10/20
3. Hanson, T.D.: uthash: a hash table for c structures, `https://troydhanson.github.io/uthash/`, last accessed on 26/09/20
4. Hanson, T.D.: utstack: intrusive stack macros for c, `https://troydhanson.github.io/uthash/utstack.html`, last accessed on 25/04/21
5. Paxson, V.: Lexical analysis with flex, for flex 2.6.2, `https://westes.github.io/flex/manual/`, last accessed on 08/09/20
6. Santos, L., Nalon, C.: Tac - the three address code interpreter, `https://github.com/lhsantos/tac`, last accessed on 06/05/21

Appendix A

Regex

```
DIGIT           : [0-9] ;
NDIGIT          : [1-9] ;
LETTER          : [a-zA-Z] ;
WHITESPACE      : [ \t]+ ;
NEWLINE         : \n ;
ADD             : "+" ;
SUB             : "-" ;
MULT            : "*" ;
DIV             : "/" ;
ASSIGN          : "=" ;
PARENT_LEFT     : "(" ;
PARENT_RIGHT    : ")" ;
BRACK_LEFT      : "{" ;
BRACK_RIGHT     : "}" ;
SEMICOLON       : ";" ;
COMMA           : "," ;
OR_OP           : "||" ;
AND_OP          : "&&" ;
NOT_OP          : "!" ;
EQ_OP           : "==" ;
GE_OP           : ">=" ;
LE_OP           : "<=" ;
NE_OP           : "!=" ;
G_OP            : ">" ;
L_OP            : "<" ;
ID              : ({LETTER}|"_")({LETTER}|{DIGIT}|"_")* ;
STRING          : \"([^(\"\')])*\" ;
CHAR            : (\'(.|\\a|\\b|\\f|\\n|\\r|\\t|\\v|\\\\|\\\'
                              |\\\"|\\\?)\');
INTEGER         : {NDIGIT}{DIGIT}*
                | "0" ;
FLOAT           : {DIGIT}+\.{DIGIT}+ ;
TYPE            : "int"
                | "float"
                | "elem"
                | "set" ;
IF              : "if" ;
ELSE            : "else" ;
FOR             : "for" ;
FORALL          : "forall" ;
RETURN          : "return" ;
```

```
READ            : "read" ;
WRITE           : "write" ;
WRITELN         : "writeln" ;
IN              : "in" ;
IS_SET          : "is_set" ;
ADD_SET         : "add" ;
REMOVE          : "remove" ;
EXISTS          : "exists" ;
EMPTY           : "EMPTY" ;
COMMENT         : "//".* ;
```

Appendix B

Grammar

```
program : stmts
        ;

stmts   : stmts stmt
        | stmt
        ;

stmt    : func_stmt
        | var_decl_stmt
        ;

func_stmt    : TYPE ID PARENT_LEFT param_list PARENT_RIGHT
                 compound_block_stmt
             ;

var_decl_stmt    : TYPE ID SEMICOLON
                 ;

param_list   : param_list COMMA TYPE ID
             | TYPE ID
             | /* empty */
             ;

simple_param_list    : simple_param_list COMMA simple_expr
                     | simple_expr
                     | /* empty */
                     ;

compound_block_stmt  : BRACK_LEFT block_stmts BRACK_RIGHT
                     | BRACK_LEFT BRACK_RIGHT
                     ;

block_stmts  : block_stmts block_item
             | block_item
             ;

block_item   : var_decl_stmt
             | block_stmt
             ;

block_stmt   : compound_block_stmt
```

```
             | func_call SEMICOLON
             | set_func_call SEMICOLON
             | flow_control
             | READ PARENT_LEFT ID PARENT_RIGHT SEMICOLON
             | WRITE PARENT_LEFT simple_expr PARENT_RIGHT
               SEMICOLON
             | WRITELN PARENT_LEFT simple_expr PARENT_RIGHT
               SEMICOLON
             | ID ASSIGN simple_expr SEMICOLON
             | RETURN simple_expr SEMICOLON
             ;

flow_control_if : IF PARENT_LEFT
                ;

flow_control    : flow_control_if or_cond_expr PARENT_RIGHT
                  block_item %prec THEN
                | flow_control_if or_cond_expr PARENT_RIGHT
                  block_item ELSE block_item
                | FORALL PARENT_LEFT set_expr PARENT_RIGHT
                  block_item
                | FOR PARENT_LEFT opt_param opt_param
                  PARENT_RIGHT block_item
                | FOR PARENT_LEFT opt_param opt_param
                  for_expression PARENT_RIGHT block_item
                ;

opt_param    : SEMICOLON
             | for_expression SEMICOLON
             ;

for_expression  : decl_or_cond_expr
                | for_expression COMMA decl_or_cond_expr
                ;

decl_or_cond_expr   : or_cond_expr
                    | TYPE ID ASSIGN simple_expr
                    | ID ASSIGN simple_expr
                    ;

or_cond_expr    : or_cond_expr OR_OP and_cond_expr
                | and_cond_expr
                ;

and_cond_expr   : and_cond_expr AND_OP unary_cond_expr
                | unary_cond_expr
                ;

unary_cond_expr : NOT_OP unary_cond_expr
                | eq_cond_expr
```

```
                 ;

eq_cond_expr     : eq_cond_expr equal_ops rel_cond_expr
                 | rel_cond_expr
                 ;

equal_ops    : EQ_OP
             | NE_OP
             ;

rel_cond_expr    : rel_cond_expr rel_ops rel_cond_stmt
                 | rel_cond_stmt
                 ;

rel_cond_stmt    : arith_expr
                 | EMPTY
                 ;

rel_ops : L_OP
        | G_OP
        | LE_OP
        | GE_OP
        | IN
        ;

set_expr     : simple_expr IN simple_expr
             ;

func_call    : ID PARENT_LEFT simple_param_list PARENT_RIGHT
             ;

set_func_call    : IS_SET PARENT_LEFT ID PARENT_RIGHT
                 | ADD_SET PARENT_LEFT set_expr PARENT_RIGHT
                 | REMOVE PARENT_LEFT set_expr PARENT_RIGHT
                 | EXISTS PARENT_LEFT set_expr PARENT_RIGHT
                 ;

simple_expr : arith_expr
            | func_cte_expr
            ;

func_cte_expr    : EMPTY
                 | STRING
                 | CHAR
                 ;

func_expr        : func_call
                 | set_func_call
                 | PARENT_LEFT func_cte_expr PARENT_RIGHT
                 ;
```

```
arith_expr   : arith_expr ADD term
             | arith_expr SUB term
             | term
             ;

term     : term MULT mid_factor
         | term DIV mid_factor
         | mid_factor
         ;

mid_factor   : SUB factor %prec UMINUS
             | factor
             ;

factor   : INTEGER
         | FLOAT
         | ID
         | PARENT_LEFT arith_expr PARENT_RIGHT
         | func_expr
         ;
```