# 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's a collection of elements that follows a certain property, and with them, it's 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's used to perform logical operations that are fundamental to the computer's existence itself.

This language aims to extend 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 Annex 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 grammar defined in Annex 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

Tokens 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 doesn't 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 and an instance of an internal object from *uthash* UT_hash_handle, called hh.

There are four main functions to help add, delete and find words in the symbol table, void add_word(int key, char *name), struct word *find_word(int word_key), 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 contain only the variables positions and names, e.g. the statement "`int var;`" will be initialized in the position 0 with name "var".

### 2.4 Code Structure and Custom Functions

GCC, version 9.3.0, and FLEX [4] 2.6.4 were 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.

### 2.5 Tests

The instructions to compile and run *c7* lexer are in the file *README.md*. After executing the usage steps, there are some files to test the lexical cases. The files "src/tests/lexer/valid_1.c7" and "src/tests/lexer/valid_2.c7" showcase valid patterns. The files "src/tests/lexer/invalid_1.c7" and "src/tests/lexer/invalid_2.c7" shows some invalid patterns that are not understood by the language. There are three errors in the first file, in lines 3, 5, and 6, and two in the second file, both in line 6.

## 3 Syntax Analyzer

### 3.1 Architecture

The syntax analyzer in *c7* is called PARSER. It is a module that receives 2 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 identifies, e.g., type and value casting, and the parser also generates the abstract syntax tree. The overview of its architecture can be found in Figure 1.

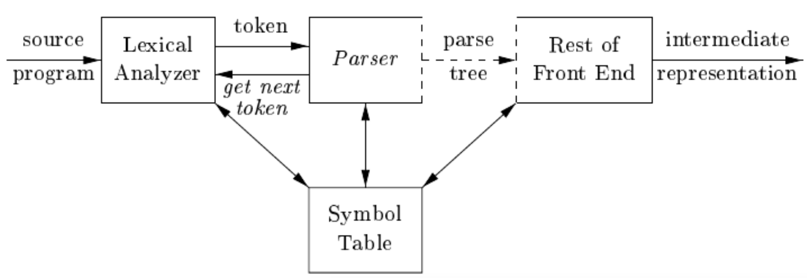


**Fig. 1.** *c7* parser inputs and outputs, image from [1].

The process of parsing an instruction goes from trying to find a possible syntax match of the tokens received from the lexer, starting from the initial symbol in the grammar, until it finds the correct patterns, otherwise an error is emitted. 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 grammar defined, 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 doesn't 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 Abstract Syntax Tree

The AST (Abstract Syntax Tree) is generated by the parser using the grammar rules (see syntax grammar in Annex 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**.

### 3.5 Tests

The instructions to compile and run *c7* parser are in the file *README.md*. After executing the usage steps, there are some files to test the parser cases. The files "src/tests/parser/valid_1.c7" and "src/tests/parser/valid_2.c7" showcase valid patterns. The files "src/tests/parser/invalid_1.c7" and "src/tests/parser/invalid_2.c7" shows some invalid patterns that are not understood by the language. There are four errors in the first file, in lines 5, 8, and two in line 16, and there are two errors in the second file, line 4 and 6.

## 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. Paxson, V.: Lexical analysis with flex, for flex 2.6.2, `https://westes.github.io/flex/manual/`, last accessed on 08/09/20

# 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             : \'.?\' ;
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 ID
                     | ID
                     | /* empty */
                     ;

flex_block_struct    : compound_block_stmt
                     | block_stmt
                     ;

compound_block_stmt : BRACK_LEFT block_stmts BRACK_RIGHT
                    | BRACK_LEFT BRACK_RIGHT
                    ;

block_stmts : block_stmts block_stmt
            | block_stmt
            ;

block_stmt  : var_decl_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 PARENT_LEFT or_cond_expr PARENT_RIGHT
                              flex_block_struct
                | IF PARENT_LEFT or_cond_expr PARENT_RIGHT
                  flex_block_struct ELSE flex_block_struct
                | FORALL PARENT_LEFT set_expr PARENT_RIGHT
                  flex_block_struct
                | FOR PARENT_LEFT opt_param opt_param
                                                PARENT_RIGHT
                  flex_block_struct
                | FOR PARENT_LEFT opt_param opt_param
                                                for_expression
                  PARENT_RIGHT flex_block_struct
                ;

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
                | func_expr
                ;

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_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 factor
         | term DIV factor
         | factor
         ;

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