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.

Later in the architecture, a symbol table structure is going to be added to be used to store the **ID**'s *lexemes* and their additional information, such as value, type, etc; for example, the id *var* presented above will be stored in the symbol table in the position zero, with the value 'var' and type 'int' and the parser can rescue this information to proceed with the compiler actions for syntax and semantics procedures.

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 Code Structure and Custom Functions

GCC, version 9.3.0, and FLEX [2] 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*>*);* };". The function *void handle_token(int token)* is a switcher that is responsible to show the valid tokens, 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.

2.4 Tests

The instructions to compile and run *c7* 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 2, 3, and 4, and two in the second file, both in line 4.

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. 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

```
nzerodigit    := '1'  | ... | '8' | '9'
digit         := '0' | nzerodigit
letter        := 'a' | ... | 'z' | 'A' | ... | 'Z'
whitespace    := ' ' | '\t'
newline       := '\n'
add           := '+'
sub           := '-'
mult          := '*'
div           := '/'
operator      := add | sub | mult | div
assign        := '='
parent_left   := '('
parent_right  := ')'
brack_left    := '{'
brack_right   := '}'
semicolon     := ';'
comma         := ','
or_op         := '||'
and_op        := '&&'
not_op        := '!'
boolean_op    := '==' | '>='| '<=' | '!=' | '>' | '<' | '<>' |
                          '~'
delimiter     := '.' | ':' | '[' | ']'
other_s       := '%' | '?' | '_' | '\' | '^' | '@' | '`'
symbols       := whitespace | newline | operator | boolean_op
                            | delimiter | parent_left |
                            parent_right | brack_left |
                            brack_left | semicolon | comma
                             | or_op | and_op | not_op |
                            other_s
string       := '"' name_dq '"' | "'" name_sq "'"
name_dq      := "" | char_dq name_dq
char_dq      := letter | digit | symbols | "'"
char         := "'" (name_sq)? "'"
name_sq      := letter | digit | symbols
integer      := nzerodigit digit* | '0'
float        := [digit]+"."digit+
number       := integer | float
id           := (letter | '_') (letter | digit | '   _')*
type         := 'int' | 'float' | 'elem' | 'set'
if           := 'if'
```

```
else          := 'else'
for           := 'for'
forall        := 'forall'
return        := 'return'
read          := 'read'
write         := 'write'
writeln       := 'writenln'
in            := 'in'
empty         := 'EMPTY'
comment       := '//' symbols*
```

Appendix B

Grammar

The terminal *EMPTY_STRING* is also well known in the literature by ε.

```
# general
program      := block
block        := BRACK_LEFT stmts BRACK_RIGHT
stmts        := stmts stmt
              | EMPTY_STRING
stmt         := READ PARENT_LEFT ID PARENT_RIGHT
              | WRITE PARENT_LEFT expr PARENT_RIGHT
              | WRITELN PARENT_LEFT expr PARENT_RIGHT
              | cond_expr ASSIGN expr SEMICOLON
              | matched_stmt
              | open_stmt
              | ID PARENT_LEFT param_list PARENT_RIGHT
              | FOR PARENT_LEFT optexpr SEMICOLON optexpr
                                          SEMICOLON optexpr
                                           PARENT_RIGHT
                                          stmt
              | FORALL PARENT_LEFT ID IN ID PARENT_RIGHT stmt
              | TYPE ID PARENT_LEFT param_list PARENT_RIGHT
                                          stmt
              | RETURN expr SEMICOLON
              | COMMENT
              | block
optexpr      := EMPTY_STRING
              | expr
matched_stmt := IF PARENT_LEFT expr PARENT_RIGHT matched_stmt
                             ELSE matched_stmt
              | block
open_stmt    := IF PARENT_LEFT expr PARENT_RIGHT stmt
              | IF PARENT_LEFT expr PARENT_RIGHT matched_stmt
                                          ELSE open_stmt
expr         := ID PARENT_LEFT expr_list PARENT_RIGHT
              | expr OR_OP join
              | join
join         := join AND_OP cond_expr
              | cond_expr
expr_list    := expr_list , expr
              | expr
cond_expr    := cond_expr BOOLEAN_OP add_op
              | add_op
add_op       := add_op ADD term
```

```
            | add_op SUB term
            | term
term        := term MULT factor
            | term DIV factor
            | factor
factor      := PARENT_LEFT expr PARENT_RIGHT
            | NUMBER
            | NOT_OP NUMBER
            | SUB NUMBER
            | ID
            | EMPTY
param_list  := param_list COMMA parameter
            | parameter
parameter   := ID
```