# Assignment 1 – Lexical Analysis

The work should be submitted by a team of 2-3 students

Submission date: 3.12.2019

Syntax of a mini programming language is described by the following (upper case is used to show variables in grammar G):

**Grammar G**

PROGRAM → BLOCK

BLOCK → **block** DECLARATIONS **begin** STATEMENTS **end**

DECLARATIONS → DECLARATION | DECLARATION ; DECLARATIONS
DECLARATION → VAR_DECLARATION | TYPE_DECLARATION
VAR_DECLARATION → id : SIMPLE_TYPE |
id : **array** [SIZE] **of** SIMPLE_TYPE |
id : type_name
SIZE → int_num
SIMPLE_TYPE → **integer** | **real**

TYPE_DECLARATION → **type** type_name **is** TYPE_INDICATOR
TYPE_INDICATOR → ENUM_TYPE | STRUCTURE_TYPE

ENUM_TYPE → **enum** { ID_LIST }
ID_LIST → id | id , ID_LIST

STRUCTURE_TYPE → **struct** { FIELDS }
FIELDS → FIELDS; FIELD | FIELD
FIELD → VAR_ DECLARATION

STATEMENTS → STATEMENT | STATEMENTS; STATEMENT
STATEMENT →
VAR_ELEMENT = EXPRESSION |
**switch** (KEY) { CASE_LIST; **default** : STATEMENTS } |
**break** |
BLOCK

VAR_ELEMENT → id | id[EXPRESSION] | FIELD_ACCESS

KEY → VAR_ELEMENT
CASE_LIST → CASE | CASE_LIST CASE
CASE → **case** KEY_VALUE : { STATEMENTS }
KEY_VALUE → int_num | id

FIELD_ACCESS → id.id | id.FIELD_ACCESS

EXPRESSION → SIMPLE_EXPRESSION | SIMPLE_EXPRESSION ar_op EXPRESSION
SIMPLE_EXPRESSION → int_num | real_num | VAR_ELEMENT

## Tokens

Below, the various groups of tokens existing in the language are listed. Such grouping is convenient for user of the language: it helps to understand the basic elements (building blocks) of the language.

**BUT**: for construction of a compiler, each operation, each keyword, and each separation sign should be implemented as a token of a different kind.

int_num : unsigned integer number (e.g. 0 , 468)
real_num : unsigned fixed-point real number (e.g. 0.75 , 34.086)

ar_op : binary arithmetic operation (addition, subtraction, multiplication, division)

assignment =

. dot - used for access to fields of structures

id - as usual, may contain letters (lower and upper case) and digits
- may contain underscores (קו תחתון) ; e.g. a1_c23_e4_56
- id can only start with letter
- id can not end with underscore
- several underscores can not appear one after another (e.g. ab___cd is not a legal id)

type_name - defined following same rules as id (see above), except that it must start with underscores (קו תחתון) for example:
_color - this is a legal type_name, but it is not a legal id
salary - this is a legal id, but it is not a legal type_name

keywords of the language : are shown in bold

separation signs: colon ,
semicolon ;
parentheses ( )
brackets [ ]
curly braces { }

## Comments

A comment starts with $$ and lasts till the end of the line

## Stage 1 of the project - Lexical analysis

1. Implement lexical analyzer (using FLEX), as follows:
   - Lexical analyzer reads text from the input file and identifies tokens. This happens when function next_token() is called.
   - When a token is identified in the input text, it should be stored in a data structure. For each token, the following attributes are saved:
     * token's kind
     * token's lexeme
     * number of the line in the input text in which this token was found.

     This is done by calling the function create_and_store_token with the relevant three parameters
   - Blanks, tabs, new lines, comments – are not tokens, and should be ignored
   - For each token, print (on a separate line) its kind (e.g. rel_op , int_num , etc.) and lexeme
   - Each operation, keyword, separation sign and each type of number should be implemented as a token of a different kind
   - Kinds of tokens are coded with integer numbers, for example:

     ```
     # define  ID_tok  1
     # define COMMA_tok  2
     ```

2. Error handling:
   - Lexical errors: each time the lexical analyzer finds a symbol that doesn't start any legal token, it sends an appropriate message
   - Each error message includes
     - information on the relevant line number (so that the user can easily locate the place in input where the error occurs)
     - the letter that doesn't start any token.

## Structure of implementation:

- a file with FLEX definitions (from which the tool will generate LEXYY.c); it contains:
  * regular expressions that describe tokens of the language;
  * actions the that lexical analyzer should perform when it identifies tokens in the input text (creation and storage of the token by calling create_and_store_token)
- .H file containing token definitions (token structure, list of token kinds)

## Submission

On the course site, a separate detailed document will be published, that describe:

- Development instructions: which operating systems and compilers can be used to implement the project
- Files (sources, executable, etc.) to be submitted