**Assignment 1 – Lexical Analysis**

Submission: by a team of 2 students

Submission date: 18 April, 2019

Syntax of a mini programming language is described by the following grammar (note that upper case is used to show variables in the grammar, and keywords of the language are shown in bold).

**Grammar G**

PROGRAM 🡪 **program** DEFINITIONS; STATEMENTS **end**

DEFINITIONS 🡪 VAR\_DEFINITIONS; FUNC\_DEFINITIONS

VAR\_DEFINITIONS 🡪 VAR\_DEFINITION | VAR\_DEFINITION; VAR\_DEFINITIONS

VAR\_DEFINITION 🡪 TYPE VARIABLES\_LIST

TYPE 🡪 **real** | **integer**

VARIABLES\_LIST 🡪 VARIABLE | VARIABLES\_LIST , VARIABLE

VARIABLE 🡪 id | id [int\_number] /\* id is a variable name \*/

FUNC\_DEFINITIONS 🡪 FUNC\_DEFINITION | FUNC\_DEFINITIONS FUNC\_DEFINITION

FUNC\_DEFINITION 🡪 RETURNED\_TYPE id (PARAM\_DEFINITIONS) BLOCK

/\* id is a function name \*/

RETURNED\_TYPE 🡪 **void** | TYPE

PARAM\_DEFINITIONS 🡪 ε | VAR\_DEFINITIONS

STATEMENTS 🡪 STATEMENT; | STATEMENT; STATEMENTS

STATEMENT 🡪 VARIABLE = EXPRESSION |

BLOCK |

**return** | **return** EXPRESSION |

FUNCTION\_CALL

BLOCK 🡪 **{** VAR\_DEFINITIONS; STATEMENTS **}**

FUNCTION\_CALL 🡪 id (PARAMETERS\_LIST) /\* id is a function name \*/

PARAMETERS\_LIST 🡪 ε | VARIABLES\_LIST

EXPRESSION 🡪 int\_number | real\_number | VARIABLE | id ar\_op EXPRESSION

**Tokens**

As usual in programming languages, in grammar G there are several groups of terminals:

IDs (names of variables and functions):

- may contain letters (lower and upper case) and digits

- can start only with a letter

- may contain underscores (קו תחתון); for example: ab\_cd\_ef

- several underscores can not appear one after another (for example ab\_\_\_cd is

not a legal id)

- id can not end with underscore (for example ab\_ is illegal)

Operatios:

ar\_op : arithmetic operations multiplication (\*) and division (/)

assignment operation =

Numbers:

int\_number : unsigned integer number (e.g. 2019)

real\_numer : unsigned real number in the fixed point format (e.g. 2.78)

Note:

* in both types of numbers, leading zeros (אפסים מובילים) are not allowed; so that 007 and 01.23 are both illegal
* on the other side, the integer number 0 is legal, as well as real numbers in which integer part is 0 – such as 0.15

Keywords:

In the grammar, they are shown in bold

Separation signs:

Comma, semicolon, and two types of parentheses: [ ] , { }

**IMPORTANT NOTE:** Each operation, each keyword, each separation sign and each type of number should be implemented as a token of a different kind. The reason for this requirement is explained in a separate document by name Tips for Lexical Analysis.

**Comments**

At every place in the program, a comment can be placed. Similar to C++, a comment starts with -- , and extends 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 yylex() is called (it is automatically created by FLEX from

definitions that you supply to this tool).

- 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 by name

create\_and\_store\_token

with the relevant three parameters. The function is a part of the package called

Token Storage supplied on the course site (as a .rar file) with this task.

- Blanks, tabs, new lines, comments – are not tokens, and should be ignored

- Kinds of tokens are coded using enumeration, for example:

enum tok\_kind (ID, COMMA, INT\_NUM, ….)

- For each token, print on a separate line its kind (e.g. INT\_NUM , etc.)

and lexeme

1. Error handling:

- Lexical errors: each time the lexical analyzer finds a symbol that doesn't start

any legal token, it sends an appropriate message (that is printed on a separate line)

- 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 that the lexical analyzer should perform when it identifies tokens in the

input text (creation and storage of tokens 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 (sourses, executable, etc.) to be submitted

**Instructions on development environments and format of messages**

These are published in a separate file, along with this assignment.