## Appendix I: Toyger Lexical Specification

Toyger tokens include **keywords, punctuation elements, operators, IDs, integers, strings,** and **comments**. They are defined as the following:

- **Keywords:** `let in end var function printint printstring getint return if then else for to do int string void`
- **Punctuation elements:** `( ) : , = ;`
- **Operators:**
  - Arithmetic: :=(assignment) `+ - * /`
  - Comparison: `==` (equality) `< <= > >=` `<>` (not equal)
- **Identifiers:** any non-empty sequence of letters (a-z and A-Z), digits (0-9), and underscores(_) starting with a letter or an underscore, and is not a keyword.
- **Numbers** (integers only)**:** any non-empty sequence of digits (0-9) with no redundant leading zeros.
- **Strings:** a possibly empty sequence of characters between (and including) the closest pair of quotation marks ("). A string cannot span more than one line. A literal quotation mark can be included as part of a string using the C-style escape format (\").
- **Comments:** start with double slash `//` until the end of the line or the end of the input (`//` in strings loses its special meaning).

**Note:**

- Toyger is case-sensitive.
- White-spaces (space, tab, newline) are allowed in input and work as delimiters between tokens.
- For comments, your scanner should recognize them but not report them to the parser -- similar to what the scanner needs to do for white-spaces.

## Appendix II: Toyger Syntax Specification

A Toyger program consists of declarations followed by a sequence of statements. **Notes**:

- In the grammar below, operators and punctuation elements are used directly (e.g. : and = ).
- Keywords are underlined.
- Other token names are capitalized (e.g. ID).
- `program` is the start symbol of the grammar.
- Spaces shown in grammar rules are not required from the input – they are inserted to make the productions easier to read.
- Comment is not included in the grammar since it can be placed between any two legal tokens. Your scanner should recognize them but there is no need to report them to the parser.

```
program        → let decs in statements end
decs           → dec decs | ϵ
dec            → var_dec | function_dec
var_dec        → var ID : type
type           → int | string | void
function_dec   → function ID (params):type = local_dec statements end |
                 function ID ( ):type = local_dec statements end
local_dec      → let var_decs in | ϵ
var_decs       → var_decs var_dec | ϵ
params         → params , parameter | parameter
parameter      → ID : type
statements     → statements ; statement | statement
statement      → assignment_stmt | print_stmt | input_stmt |
                 if_stmt | for_stmt | call_stmt | return_stmt
assignment_stmt→ ID := expr
input_stmt     → ID := getint()
return_stmt    → return expr  | return
call_stmt      → ID ( ) |  ID ( expr_list )
print_stmt     → printint ( expr ) | printstring ( expr )
rel_expr       → expr == expr |  expr <> expr |  expr < expr | expr <= expr|
                 expr > expr |  expr >= expr
if_stmt        → if (rel_expr) then statements end |
                 if (rel_expr) then statements else statements end
for_stmt       → for ID := expr to expr do statements end
expr           → expr + term | expr – term | term
term           → term * factor | term / factor | factor
factor         → ( expr ) | NUMBER | STRING | ID | call_stmt
expr_list      → expr_list , expr | expr
```

# Appendix III: Toyger Semantic Rules

## Scoping

A Toyger program consists of function/variable declarations followed by a sequence of statements. Every source file is associated with a global scope. Global names include all function names and variable names declared outside of functions. Each function introduces a separate local scope that is nested inside the global scope. Identifiers belongs to a local scope includes parameters and local variables of that function. No nested function definitions are allowed. Toyger uses static scoping.

- Global names (functions/variables) are visible after their declarations until the end of the file.
    - The sequence of statements after declarations can access all global names.
    - Global names that have been declared are visible inside functions unless they are hidden by a local variable with the same name. This implies that recursive functions are allowed but we do not support mutual recursion.
- Local names (variables only) are visible only within the function in which they are declared or introduced as parameters.

## ID Definitions

Toyger IDs need explicit declarations. No names can be used before they are declared.

- A function cannot be called before it has been defined.
- A variable cannot be referenced if it has not been declared or introduced as a parameter.
- Each name/ID can only be declared once *in each scope*:
    - It is not allowed to define multiple functions with the same name (even if they have different return types and/or different parameters).
    - It is not allowed to define a function and a global variable with the same name.
    - It is not allowed to declare a variable multiple times in one scope.
        - For a function, we cannot have a local variable and a parameter sharing the same name.
    - It is allowed to define a local variable with the same name as a global variable or function. The global name will be shadowed by the local name.

## Types

Constants.

- Token NUMBER is of type integer
- Token STRING is of type string.

Variables. Variables in Toyger can take two possible types: integer and string.

- A global/local variable is declared to take a type.
- A parameter gets its type specified in the parameter list of a function.

Functions. Function declaration defines the name, return type, and the list of parameters of a function. To simplify the task, for this project, you will not need to perform any type checking for function calls (neither for the arguments nor for the returns). For function calls in expressions, you can assume that the return type is always integer.

**Special Note for Project 4**:

- All variables/parameters are of integer type;

- Functions take no more than four parameters;
- Function return (if any) must be integer type;

**Instructions**

The semantics of most instructions should be straightforward. The clarification for the loop (for_stmt) is as below:

```
for_stmt        →  for ID := expr1 to expr2 do statements end
```

Semantic as pseudocode using a while loop:

```
ID := expr1;// evaluating expr1, initializing loop variable ID
while (ID <= expr2){ //evaluating expr2, comparison
     statements; //loop body
     ID := ID+1;   //updating loop variable ID
}
```