Assignment 1 – Lexical Analysis

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

Submission date: 16.4.2020, 23:55

Below, syntax of a mini programming language is described (upper case is used to show variables in grammar G):

Grammar G

PROG → GLOBAL_VARS FUNC_PREDEFS FUNC_FULL_DEFS

GLOBAL_VARS → GLOBAL_VARS VAR_DEC | VAR_DEC /* declarations of global variables */

VAR_DEC → TYPE id ; | TYPE id [DIM_SIZES] ; /* allow multi-dimensional arrays */

TYPE → **int** | **float** /* variables can be only of these types */

DIM_SIZES → int_num | int_num , DIM_SIZES /* list of sizes in each of the dimensions */

FUNC_PREDEFS → FUNC_PREDEFS FUNC_PROTOTYPE; | FUNC_PROTOTYPE;

FUNC_PROTOTYPE → RETURNED_TYPE id (PARAMS)

FUNC_FULL_DEFS → FUNC_WITH_BODY FUNC_FULL_DEFS | FUNC_WITH_BODY

FUNC_WITH_BODY → FUNC_PROTOTYPE COMP_STMT

RETURNED_TYPE → TYPE | **void**

PARAMS → PARAM_LIST | ε /* function can be without parameters */

PARAM_LIST → PARAM_LIST , PARAM | PARAM

PARAM → TYPE id | TYPE id [DIM_SIZES]

COMP_STMT → { VAR_DEC_LIST STMT_LIST } /* if VAR_DEC_LIST is non-empty, then COMP_STM is in fact a block that contains declarations of local variables. Otherwise it is just a grouped series of statements */

VAR_DEC_LIST → VAR_DEC_LIST VAR_DEC | ε

STMT_LIST → STMT_LIST ; STMT | STMT

STMT → VAR = EXPR | COMP_STMT | IF_STMT | CALL | RETURN_STMT

/* note that in the assignment, the left hand side can be either a simple variable, or an array element – see definition of VAR below */

IF_STMT → **if** (CONDITION) STMT

/* note that STMT can be a COMP_STMT, thus allowing execution of any amount of statements when condition is True */

CALL → id (ARGS)

ARGS → ARG_LIST | ε

ARG_LIST → ARG_LIST , EXPR | EXPR

RETURN_STMT → **return** | **return** EXPR

VAR → id | id [EXPR_LIST] /* to allow access to multi-dimensional arrays */

EXPR_LIST → EXPR ,_LIST EXPR | EXPR

CONDITION → EXPR rel_op EXPR

EXPR → EXPR + TERM | TERM

TERM → TERM * FACTOR | FACTOR

FACTOR → VAR | CALL | int_ num | float_num | (EXPR)

Tokens

Below, the various groups (but not kinds) 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.

Numbers

int_num : unsigned integer number (e.g. 2020 , 27)

float_num : unsigned floating-point real number; its presentation must include exponent whose value is an integer number with or without sign (e.g. 75e5 , 34.86e-3 , 2.78e+10)

Operations

ar_op : binary arithmetic operation (in this language – only addition or multiplication)

rel_op: comparison operations < , <= , == , >= , > , !=

assignment_op : this is the assignment operation = (not a comparison operation)

Identifiers

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 a lower-case letter
- id can not end with underscore
- several underscores can not appear one after another (e.g. ab___cd is not a legal id)

Keywords

In this language : int, float, void, if, return (in the grammar they are shown in bold)

Separation signs

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

Comments

A comment starts with /* and ends with */ (as in C); it can occupy several lines

==

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. COMMA_tok , ID_tok , 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 using enumeration, or using 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 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 describes:

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