Assignment 2 – Syntax Analysis

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

Submission date: 21.5.2020, 23:55

Assignment 1 was devoted to development of lexical analysis for the language that is described by the grammar presented below.

<u>In Assignment 2 the goal is to develop a parser (syntax analyzer) for this language, according to the Recursive Descent Parsing method.</u>

This assignment is an incremental step in the project development: it is based on what was developed in Assignment 1.

<u>Grammar G</u>

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~~ 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)

Program (according to the grammar G)

A program is a series of declarations:

- Global variables
- Function pre-declarations.
  Pre-declaration (or in other words, prototype) specifies the function's name and signature.
- Full declarations of functions.
  Full declaration contains both the function's prototype and body that implements the function.

Example:

```
int glob_var1;  float glob_var2;

int func1 (int a, int b);

void main()

        { int x; global_var1 = func2(1, 10) + y; return }

int func2 (int a, int b)

        { return (a+b) }
```

Note:

- this program contains some errors; for example: variable y is not declared, func2 is called before it is declared
- **however, it is syntactically correct:** it can be derived in the grammar G, and hence parser should successfully accept it
- the mentioned errors are so called semantic errors; they will be addressed in stage 3 of the project, devoted to semantic analysis

Scope of declaration

Note that variables can be declared on the program level (global variables) and in blocks (local variables).

In this grammar, block is described by COMP_STMT. A block starts with declarations of variables, followed by a series of statements. A statement can itself be a block (i.e. a COMP_STMT). This means that nested blocks are allowed, with their local variables.

As opposed to variables, functions can be declared only on the program level, but not in blocks (in other words, local functions are not allowed).

Declaration of variables

A variable in the language may have one of the following types:

- simple type (integer, float)
- array of a simple type; note that multi-dimensional arrays are allowed

Example:

```
int grade ;
float average weight ;
int matrix{10,20,20];
float linear_array [20] ;
```

Note that in an array, the size of each dimension is specified by an explicit integer number.

Expressions

In the language, there are two kinds of expressions:

- Arithmetic expressions, described in the grammar by EXPR
- Boolean expressions, described in the grammar by CONDITION

Arithmetic expression EXPR allow only two kinds of operations: addition and multiplication. These operations can be applied to numbers, ids, array elements and function calls. Parentheses can be also used to build complex expressions from simpler ones.

Example:

```
x * f(y+z, 5) + (arr[3] * 4.35e-2)
```

Note the way arithmetic expressions are defined using the concepts TERM and FACTOR: this enforces the correct order of operations even when no parentheses are used. In particular, according to the grammar, it is guaranteed that in the above example the order of operations will be as follows:

$$\underbrace{\underbrace{x * f(y+z, 5)} + \underbrace{(arr[3] * 4.35e\text{-}2)}}$$

Boolean expression CONDITION is obtained by applying a relational (comparison) operation to two arithmetic operations.

Example:

```
x * f(y+z, 5)  !=  (arr[3] * 4.35e-2)
```

Statements

The allowed kinds of statements:

- assignment (note that the left-hand side can be either id, or array element)
- conditional statement (IF_STMT)
- function call
- return
- block (COMP_STMT)

Example:

```
x = func(arr[i+1], y);

arr[5] = 6.7e+3;

if ( x >= y) {int local_var;  local_var = x * 67} ;

merge_function(arra1 , arr2);

return (arr[10] + 2.3e-2)
```

==

TASKS

Before writing code

1. Eliminate left recursion and common left prefixes in the given grammar (both direct and indirect).

2. For the obtained grammar, perform calculation of attributes Nullable, First and Follow for each of the grammar's variables.

   This information is needed for:

   - Writing code of parser's functions
   - Implementation of recovery from syntax errors in these functions.

Coding

3. Implement service functions:

   next_token()

   back_token()

   match()

   Note that implementation of next_token() and back_token() is based on the use the data structure for tokens storage, that was supplied to you in Assignment 1 (see the package by name Token Storage).

4. Implement parser that performs Recursive Descent syntax analysis.
   - All functions that implement the parser have to be placed together in a separate file. This includes:
     - a separate function for every variable in the grammar
     - function parse() , which is parser's main function
   - Activation of parser: done by calling to parser() in function `main` in the file with FLEX definitions .

5. Error handling:
   - Each time the parser gets an unexpected token, it should send an appropriate error message, saying:
     - what was the expected token/tokens
     - what is the actual token
     - in which line the error was found (so that the user can easily localize the place in input where the error occurs).
   - In addition, parser should perform a recovery (התאוששות משגיאה) and continue syntax analysis.

     Implement the recovery policy discussed in the course: in case of an error found in function parse_X() , skip tokens till arriving to either EOF or to a token in Follow(X).

6. Output of the parser is a report that contains:
   - **Sequence of derivation rules in G used during syntax analysis of the input**.

     Each used derivation rule is reported in a readable form, exactly as it appears in the grammar that was obtained after elimination of left recursion and of common left prefixes.
   - **Error messages**.

     The exact format is specified in the updated instructions document on the site of the course in MOODLE; this document is published together with this one.

All outputs produced during the execution should be recorded in an output file.