Assignment 2 – Syntax Analysis

Submission date: 5.1.2020

Assignment 1 was devoted to development of lexical analysis for the language that is described by the grammar presented below. **In Assignment 2 the goal is to develop parser (syntax analyzer) for this language.**

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

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

Program

A program in the language is a block. This (and any other) block starts with declarations of variables and types, followed by a series of statements.

Note that a statement can itself be a block, framed by the keywords **block** and **end**. This means that nested blocks are allowed, with their local variables.

Declaration of types and variables

A variable in the language may have one of the following types:
- simple type (integer, real)
- array of a simple type.
- user-defined type (for enumeration and structure types)

According to the grammar, enumeration and structure types should first be defined by a type declaration, and only then they can be used to declare variables.

For example:

```
        grade : integer;
        type _color is enum {red, blue, yellow, green, purple, brown, black, white};
        shape_color : _color;
        type _personal_data is
                struct {age : integer ; weight : real; eye_color : _color}
        my_data : _personal_data;
        a : array[20] of integer;
        arr : array[101] of real;
```

Note that size of array is specified by an explicit integer number.

Expressions

- Simple expressions are:

  numbers (e.g.: 476, 3.14)
  ids (e.g. : grade , HIT, year_2019)
  array elements (e.g.: a[5] , arr[92])
  structure fields (e.g. my_data.age)

- Simple expressions can be combined using arithmetic operations of the language to obtain more complicated expressions; for example:

  grade * my_data.age + arr[38] * 2

Statements

The allowed kinds of statements:

- assignment (note that the left-hand side can be either id, or array element, or structure field)
- switch
- break
- block

Example:

```
my_data.eye_color = brown;
a[5] = arr[20] – 15;
switch (my_data.eye_color) {
case green: {grade = 100; my_data.weight = 71.5; break}
case blue: {grade = 90; shape_color = red; break}
case brown: {grade = 80;
                 block i : integer; begin i = 1; arr[i] = grade * a[10] end; break}
default :
                  block type _error_level is enum (low, medium, high, gevald);
                           error_signal : _error_level;
                  begin
                           error_signal = medium
                  end
}
```

TASKS

Before writing code

1. Eliminate left recursion and common left prefixes in the given grammar.

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 (one function for every variable in the grammar) have to be placed together in a separate file
   - Activation of parser: done 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.

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.
   - 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.