Theory of Compilation (61304)

Semester 2018b Project

Assignment 2 – Recursive Descent Syntax Analysis

Submission date: 16.5.2019 till 23:55

Assignment 1 was devoted to development of a lexical analyzer 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 →

program VAR_DEFINITIONS; STATEMENTS **end** FUNC_DEFINITIONS

~~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

Program

A program in the language is a special kind of block (framed by the keywords **program** and **end**), followed by function definitions.
This special block starts with definitions of variables, followed by a series of commands.
Note that a command can itself be a block, framed by parenthesis { and }. This means that nested blocks are allowed, with their local variables.
Pay attention that functions can be defined on the program level only, not within blocks.

Definition of variables

A variable in the language can have either a basic type (integer, real), or be an array of a basic type. One definition may define a number of variables, for example:

```
integer a, b[20], c, d[40];
real arr[101], age;
```

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

Definition of functions

Function definition includes:

- function's name
- returned type (void, integer, real)
- list of parameters; the list can either be empty (for function with no parameters), or define parameters in the same format as the one used for definition of variables
- function's body; this is a block

Expressions

- The most simple expressions are numbers (integer number or real number).
- Another option for expression is defined as VARIABLE. This can be either a simple variable (designated by an ID), or access to array's element, e.g. a[5]. Note that according to the grammar, the index must be an explicit integer number (so that for example a[i+1] is illegal).
- Finally, expressions can be combined using arithmetic operations of the language (multiplication or division) to obtain even more complicated expressions. Note that in this case the very first argument in the expression must be an ID.

Commands

The allowed kinds of commands (in the grammar mentioned as STATEMENTS):

- assignment (note that the left-hand side can be either ID, or array's element)
- function call
- return statement
- block

<u>TASKS</u>

<u>Before coding</u>

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.

<u>Coding</u>

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.

4. Implement parser that performs Recursive Descent syntax analysis.
   - All <u>functions that implement the parser</u> (one function for every variable in the grammar) have to be placed in a separate file
   - <u>Activation of parser</u>: 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
     - 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 <u>perform a recovery</u> (התאוששות משגיאה) <u>and continue syntax analysis</u>. 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.