Formal Languages and Compilers

18 September 2020

Using the JFLEX lexer generator and the CUP parser generator, realize a JAVA program capable of recognizing and executing the programming language described in the following.

Input language

The input file is composed of two sections: *header* and *command* sections, separated by means of the sequence of characters "`###`". Comments are possible, and they are delimited by the starting sequence "`{-`" and by the ending sequence "`-}`'.

Header section: lexicon

The *header* section can contain 3 types of tokens, each terminated with the character ";":

- `<tok1>`: it begins with the character "*", followed by a word composed of an even number (at least 4) of lowercase alphabetic letters. It is then followed by an octal number (possible digits are from 0 to 7) between -37 and 123 , and optionally followed by 4 or more repetitions of the words "xx", "yy", or "zz" in any combination.
- `<tok2>`: it is composed by 2, 10, or 31 emails, where each email is a word composed of numbers, letters and characters "_" and ".", the character "@", and a word composed of letters and numbers, a ".", and the word "it", "org", or "net". Each email is separated by the character ":" or "/".
- `<tok3>`: it is the word "token3".

Header section: grammar

In the *header section* the tokens `<tok1>` and `<tok3>` can appear in **any order and number** (**even 0** times), instead, `<tok2>` can appear only **0, 1, or 2 times**.

Code section: grammar and semantic

The *command section* is composed of a list of `<commands>`. The list can be possibly **empty**, or with an **even** number of elements, **at least 4**. As a consequence, the list can be composed of 0, 4, 6, 8,... elements.

Two types of commands are possible:

- *Assignment*: it is a `<variable>` (same regular expression of C identifiers), followed by a "=", and a `<bool_expr>`. This command stores the result of the `<bool_expr>` into an entry of a global symbol table with key `<variable>`. **This symbol table is the only global data structure allowed in all the examination, and it can be written only by means of an assignment command**. Each time an *assignment* command is executed, the command prints into the screen the `<variable>` name and the associated value.
- *EQUAL*: it has the following syntax:

  `EQUAL <bool_expr> <actions_list>`

  where `<bool_expr>` represents the result of a boolean expression (i.e., a `true` or a `false` value). `<actions_list>` is a list of at least one `<action>`, where an `<action>` is the word `TO`, a `<bool_expr_a>` (same regular expression of `<bool_expr>`), the word `DO`, a `<write>` instruction, and the word `DONE`. The `<write>` instruction is the word `write`, followed by a *quoted string* and by a ";". The `<write>` instruction is executed each time the result of `<bool_expr>` equals the result of `<bool_expr_a>`.

`<bool_expr>` can contain the following logical operators: `and`, `or`, `not`, and round brackets. Operands can be `true` (the true constant), `false` (the false constant), a `<variable>` (which represents the value stored in the symbol table by an *assignment* command), and the `AND()` function. The `AND()` function takes in input a list of `<bool_expr>` separated by a ",", and returns the "logical and" of the results of the listed `<bool_expr>` (i.e., the `AND()` function returns `true` only if the results of all the listed `<bool_expr>` are `true`, otherwise it returns `false`).

Goals

The translator must execute the language, and it must produce the output reported in the example. For any detail not specified in the text, follow the example.

Example

Input:

```
token3 ;                                        {- tok3 -}
name1.surname1@skenz.it/name2.surname2@abc.net; {- tok2 -}
*abcfef-36xxyyxxyy ;                            {- tok1 -}
token3;                                         {- tok3 -}

### {- division between header and command sections -}

x1 = true;
x2 = not true and not x1 ;  {- false and false = false -}

{- AND(true, true, true, false) or false = false or false = false -}
x3 = AND(true, true, AND(true, x1), false  ) or false;


EQUAL true and false   {- true and false = false -}
TO false or false DO   {- executed -}
  write "1";
DONE
TO x1 DO      {- not executed -}
  write "2";
DONE
TO not x1 DO  {- executed -}
  write "3";
DONE
```

Output:

```
x1 true
x2 false
x3 false
"1"
"3"
```

Weights: Scanner 8/30; **Grammar** 9/30; **Semantic** 10/30