Formal Languages and Compilers

11st February 2016

Using the JFLEX lexer generator and the CUP parser generator, realize a JAVA program capable of recognizing the language described later.

Input language

The input file is divided into two sections: header and commands. The header section is separated from the commands section by means of an **even number** of "%" characters (**at least 4**) or by means of an **odd number** of "#" characters (manage with Jflex). The input file can contain C++ style comments, which start with the symbol "//" and comment all the characters until the end of the line.

The header section is composed of two types of tokens, each terminated with the character ";":

- <code>: begins with an even number between −24 and 2472. The number is followed by at least 5 characters "$" or "?" in any order and in odd number (e.g., $$$$$, $?$$$, $?$???$) or by a word composed of 4, 6 or 9 alphabetic letters.
- <date>: it is a date with the format "YYYY/MM/DD" between 2015/12/06 and 2016/03/31. Remember that the month of February has 29 days. The date is optionally followed by a hour with the format ":HH:MM" between :04:32 and :15:47.

Header section: grammar

In the header section the token <code> can appear **in any order and number** (**even 0**), instead <date> can appear **0** or **1** time.

Commands section: grammar and semantic

The commands section starts with the instruction `START`, and it is followed by a list **possibly empty** of <comands>. The instruction `START` and each <command> are ended by the character ";".

The instruction `START` sets the initial position of a *point* in a three-dimensional space. The instruction is the word `START`, followed by 3 <coordinates> separated by commas. A <coordinate> is an integer number preceded by the symbols "+" or "-".

Three commands are defined:

- **VAR { <list_of_attributes> }** <**var_name**>: saves into a global hash table an entry with key <var_name> and associated value all the information contained in <list_of_attributes>. In particular, <list_of_attributes> is a **non-empty** list of **at least 3** <attributes> separated by commas. The number of <attributes> must be a **multiple of three** (i.e., 3, 6, 9,...). Each <attribute> is composed of an <attribute_name>, the character ":" and an <attribute_value> (both, <attribute_name> and <attribute_value> must be stored for future uses). The regular expression of <var_name> and <attribute_name> is one or more alphabetic letters followed by one ore more numeric characters. <attribute_value> has the same regular expression of <coordinate>.

  The hash table with key <var_name> is the **only global variable** allowed in all the examination. **Solutions using others global variables will not be accepted.**

- **MOVE <X> <Y> <Z>**: moves the position of the point of the quantities <X>, <Y> and <Z> in the three axes (i.e., the three quantities are added, or subtracted in the case of negative values, to the previous position of the point). The three quantities can be integer numbers (with the same regular expression of <coordinate>) or a <var_name>, a dot "." and an <attribute>.

In this last case, the value related to the entry <attribute>, which is associated to <var_name>, must be extracted from the hash table and used.

- **WHEN <boolean_expression> THEN <MOVE_list> DONE**: <boolean_expression> is a boolean expression that can contain AND, OR and NOT operators in any order and number. The possible operands are numbers (which have the same regular expression of <coordinate>), or a <var_name>, a dot "." and an <attribute>. Only the operators "`==`" (equal) and "`!=`" (not equal) must be implemented for comparisons. Look the examples. If <boolean_expression> is true, the instructions listed in <MOVE_list> must be executed. <MOVE_list> is a non-empty list of only `MOVE` commands.

  Example regarding the `WHEN` command:

  ```
  // TRUE OR NOT TRUE AND FALSE = TRUE OR FALSE AND FALSE = TRUE OR FALSE = TRUE
  WHEN 1==1 OR NOT 2 != 3 AND 2 ==3 THEN
    MOVE 1, -2, 3;
  DONE
  ```

  Because <boolean_expression> is true, the three coordinates X, Y and Z of the point are updated by adding to them the values 1, -2 and 3, respectively.

Goals

The translator must execute in sequence, as reported in the example, the instruction `START` and the <commands> in the second section. Each time a `MOVE` command is executed, the translator must print the new position of the point, by adding the coordinates (argument of the command `MOVE`) to the previous position. Remember that the position cannot be stored in global variables.

Example

Input:

```
// First section: header
124$????$?;          // <code>
-12AbCd;             // <code>
2016/02/29:04:50;    // <date>
24$?$?$;             // <code>

###
// Second section: commands
START +0, +0, +0;
MOVE +1, +2, -1;   // New position: 1, 2, -1
VAR {X1: +3, Y1 : +2, Z1: +2, X2: +2, Y2:-1, Z2: +1} POS1;
MOVE +3, -2, -1;   // New position: 4, 0, -2
VAR {X1: +2, Y:+2, Z:-2} POS2;
WHEN POS1.X1 != +3 OR NOT POS2.Y == 3 THEN  // FALSE OR NOT FALSE = TRUE
  MOVE -2, +2, POS1.Z2;   // New position: 2, 2, -1
  MOVE +1, +1, +1;        // New position: 3, 3, 0
DONE;
```

Output:

```
POSITION: X=1 Y=2, Z=-1
POSITION: X=4 Y=0, Z=-2
POSITION: X=2 Y=2, Z=-1
POSITION: X=3 Y=3, Z=0
```