Formal Languages and Compilers

19 September 2016

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

Input language

The input file is composed of two sections, a *header* followed by a *states* section, separated by means of a token composed of 4 or more characters "%", the number of "%" must be even (e.g, %%%%, %%%%%%, etc.). Semantic actions are required only in the *states* section. The input file can contain C stile comments with the syntax `/* <comment> */`.

The *header* section contains 2 types of tokens, each terminated with the character ";":

- <token>: an odd number between −183 and 67, optionally followed by a word of at least 4 uppercase alphabetic letters (in an even number), followed by the word "***" or by 4 or more repetitions of the words ("xx", "xy", "yx" or "yy"), which can appear in any possible combination.
- <date>: a date with the format "YYYY/MM/DD" between 2015/09/19 and 2016/02/15. Remember that the months of September and November have 30 days. The date is optionally followed by a hour with the format ":HH:MM" between 06:13 and 15:43.

Header section: grammar

The *header* section contains one of these two possible sequences of tokens:

1. **at least** 4, and in **even** number (4, 6, 8,...) repetitions of <token>, followed by **3** or **13** repetitions of <date>
2. **three** <date> and **any number** of <token> (**even** 0). This sequence **must start** with a <date>, the second and third repetitions of <date> can be in **any position** of the sequence.

States section: grammar and semantic

The *states* section describes the evolution of *power* and *water* quantities in the International Space Station (ISS).

This section can start with the `SET` instruction. It is the word "SET" followed **one** or **two** <setters>. In the case of two <setters>, they are separated with the character "-". A <setter> is a <quantity> (i.e., the word "`POWER`" or "`WATER`") followed by a floating point number. It is not possible that the same <quantity> is repeated in the same `SET` instruction. This instruction sets the initial quantity of *power* and/or *water* to a specific value. If only one <setter> is present, or the `SET` instruction is not present, the values of *power* and/or *water* which are not explicitly specified are set to the default value 100.0. **No global variables are allowed in all the exam, as a consequence the values of the quantities `power` and `water` must be propagated and stored inside the parser stack.**

The instruction `SET`, and the two commands described later, are terminated by the character ";".

The second part of the *states* section is composed of a non-empty list of <commands>.

The two possible <commands>, which can appear in the input file **in any order**, are `STATE_CHANGE1` and `STATE_CHANGE2`. They modify the value of one of the two states variable *power* or *water*. The grammar and the semantic of the two commands is as follows:

- `STATE_CHANGE1`: Is the word "`STATE_CHANGE1`", followed by the words "`INCREASE`" or "`DECREASE`", followed by the words "`POWER`" or "`WATER`", followed by a <boolean_expression>, the character "?" and two <avg_func> (<avg_func$_T$> and <avg_func$_F$>) separated by the character ":". If <boolean_expression> is `TRUE`, the quantity of *power* or *water* is *increased* or *decreased* of a quantity equal to <avg_func$_T$>. On the contrary, if <boolean_expression> is `FALSE`, the relevant quantity is modified by a quantity equal to <avg_func$_F$>.

  <boolean_expressions> are typical boolean expressions that include the operator `AND`, `OR`, `NOT`, parenthesis and `TRUE` or `FALSE` keywords, to identify *true* and *false* boolean values, respectively. <avg_func> is the word "`AVG`", a "(", a <float_list> and a ")". <float_list> is a list, eventually **empty**, of floating point numbers or other `AVG` functions, separated by commas ",". The `AVG` function returns the mean value of the floating point numbers listed inside brackets or 0 in the case of empty list.

- `STATE_CHANGE2`: This command has the following grammar:

  `STATE_CHANGE2 <press_mod> PRESSURE <temp_mod> TEMP # <quantity> -> <var_list> ;`

  where <press_mod> and <temp_mod> are two floating point numbers, <quantity> is the word `POWER` or `WATER`, and <var_list> is a non-empty list of <variations> separated by commas ",". A <variation> is the word `PRESSURE` or `TEMP` (that identifies which of the values <press_mod> or <temp_mod> must be used), an <operation> (i.e, the words "`ADD`" or "`SUB`", which represent an addition or a subtraction, respectively), and a <value> (i.e., a floating point number). The values <press_mod> and <temp_mod> represent a change in the environment in terms of *pressure* or *temperature* that influences the state of the ISS from the point of view of *power* or *water*.

  Each <value> of a <variation> must be multiplied by the number <press_mod> or <temp_mod>, if the word `PRESSURE` or `TEMP` has been specified in the <variation>, respectively (**to this extent use inherited attributes**). The value <operation> determines, for each value included in <var_list>, if it must be added (`ADD`) or subtracted (`SUB`).

  The command `STATE_CHANGE2`, after computing the sum of all the <variations> listed in <var_list> has to update the state variable *power* if <quantity> is equal to "`POWER`", otherwise, if it is equal to `WATER`, the state variable *water* must be updated.

Goals

The translator must execute the programming language of the last section, printing for each executed command and instruction the values of the *power* and *water* state variables.

Note: in the correction scanner will be evaluated 8/30, grammar 9/30, semantic 10/30 and compilation 4/30. Total possible points are 31/30.

Example

Input:

```
/* Header section */
/* Second type of sequence. */
2015/10/17;        /* <date> */
2016/01/01:07:00;  /* <date> */
-181ABCDEF***;     /* <token> */
51xxxyxyxxyy;      /* <token */
2015/12/28:14:02;  /* <date> */
%%%%%%%%
/* States section */

/* First part: One SET instruction (power: 50.0, water: 50.0) */
SET POWER 50.0 - WATER 50.0;   /* Other examples of the SET instruction are:
                              SET WATER 50.0 - POWER 100.0;
                              SET WATER 50.0; If one quantity is empty, its default value is 100.0
                              Without a SET instruction, default value of both variables is 100.0 */

/* Second part: STATE_CHANGE1 and STATE_CHANGE2 instructions */

/* TRUE OR TRUE AND FALSE = TRUE;   AVG(2.,4.0,3.0)=3.0
   -> Power decreases of 3.0 (power: 47.0, water: 50.0) */
STATE_CHANGE1 DECREASE POWER TRUE OR TRUE AND FALSE? AVG(2.,4.0,3.0): AVG(4.0,8.);

/* NOT ( NOT TRUE OR FALSE)=NOT FALSE=TRUE;   AVG(AVG(2.0,2.0),4.0,3.0)=AVG(2.0,4.0,3.0)=3.0
   -> Water increases of 3.0 (power: 47.0, water: 53.0) */
STATE_CHANGE1 INCREASE WATER NOT ( NOT TRUE OR FALSE)? AVG(AVG(2.0,2.0), 4.0, 3.0) : AVG();

/* The result of <var_list> is: + 3.0 * 0.9 + 2.0 * 1.1 - 3.0 * 0.9 = 2.2 */
/* Power is increased of 2.2 (power: 49.2, water: 53.0) */
STATE_CHANGE2 .9 PRESSURE 1.1 TEMP # POWER -> PRESSURE ADD 3.0, TEMP ADD 2., PRESSURE SUB 3.0;
```

Output:

```
power: 50.0, water: 50.0
power: 47.0, water: 50.0
power: 47.0, water: 53.0
power: 49.2, water: 53.0
```