Formal Languages and Compilers

3 July 2015

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 *description* followed by *simulation* section, separated by means of a token composed of **3 or more** characters "$", the number of "$" must be **odd** (e.g, $$$, $$$$$, etc.). Semantic actions are required only in the *simulation* section. The input file can contain C++ stile **comments** with the syntax `//` <`comment`>.

The *description* section can contain 2 types of tokens, each terminated with the character ";":

- `<code1>`: it starts with the character "#", followed by a word with at least 4 alphabetic characters in an even number, followed by a number between −3 and 123, and optionally ended with the word IJK or the word XYZ (where the number of Z can be zero or odd: i.e., XY, XYZ, XYZZZ, XYZZZZZ, ...).
- `<code2>`: it is composed of an even number of hexadecimal numbers of 2 or 4 characters each. The hexadecimal numbers are separated by the character "-" or by the character ":".

Description section: grammar

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

1. **at least** 3, and in **odd** number (3, 5, 7,...) repetitions of `<code1>`, followed by 2 or 3 or 5 repetitions of `<code2>`
2. **two** `<code2>` and **any number** of `<code1>` (**even** 0). This sequence must start with `<code2>`, the second repetition of `<code2>` can be in any position of the sequence.

Manage these two requirements with grammar.

Simulation section: grammar and semantic

The *simulation* section describes the evolution of a colony of cells enclosed in a biosphere.

This section stats with exactly two instructions, **always present** and in **every possible order**, which initialize two state variables that are the quantity of `oxygen` and of `cells`. **No global variables are allowed in all the exam, as a consequence the values of the variables `oxygen` and `cells` must be propagated and stored inside the parser stack.**

The two instructions are the word "`OXYGEN`" or the word "`CELLS`", followed by an unsigned integer. They are used in the program to set the values of the variables `oxygen` and `cells`, respectively.

Each instruction, `OXYGEN` and `CELLS`, and the two commands described later, are terminated by the character ";".

The second part of the *simulation* 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 `MOD_STATE1` and `MOD_STATE2` that modify the value of one of the two state of variables `oxygen` or `cells`. The grammar and the semantics of the two commands is as follows:

- `MOD_STATE1`: Is the word "`MOD_STATE1`", followed by the word "`OXYGEN`" or "`CELLS`", followed by the character "-" or "+", followed by a `MAX` function. The `MAX` function is the word "`MAX`", a "(", a <int_list> and a ")". The <int_list> is a list, eventually **empty**, of unsigned integer numbers or other `MAX` functions, separated by commas ",". The `MAX` function returns the maximum of the unsigned integer numbers listed inside brackets or 0 in the case of empty list. The result of the outer `MAX` function is subtracted (in the case of "-") or added (in the case of "+") to the `oxygen` or to the `cells` state variable, in the case the word `OXYGEN` or the word `CELLS` has been specified in the command, respectively.
- `MOD_STATE2`: This command has the following grammar:

  `MOD_STATE2 TEMP <temp_mod> FOOD <food_mod> <parameter> : <list_variations> ;`

where `<temp_mod>` and `<food_mod>` are two floating point number, `<parameter>` is the word `OXIGEN` or `CELLS`, and `<list_variations>` is a non-empty list of `<variations>` separated by commas ",". A `<variation>` is a `<direction>` (i.e, the symbol "+" or "-", which represents an addition or a subtraction, respectively), a `<quantity>` (an unsigned integer number), and the word `TEMP` or `FOOD` that identifies which of the values `<temp_mod>` or `<food_mod>` must be used. The values `<temp_mod>` and `<food_mod>` represent a change in the environment in terms of temperature or food quantity that influences the cell evolution in terms of available oxygen and cells number. Each `<quantity>` of a `<variation>` must be multiplied by the value `<temp_mod>` or `<food_mod>`, if the word `TEMP` or `FOOD` has been specified in the `<variation>`, respectively (**to this extent use inherited attributes**). The value `<direction>` determines, for each value included in `<list_variations>`, if it must be added or subtracted.

The command `MOD_STATE2`, after computing the sum of all the `<variations>` listed in `<list_variations>`, must truncate the sum to an integer value, and update the state variable `oxygen` if `<parameter>` is equal to "`OXYGEN`", otherwise, if it is equal to `CELLS`, the state variable `cells` must be updated.

Goals

The translator must execute the programming language of the last section, printing for each executed command the values of the `oxygen` and `cells` state variables.

Example

Input:

```
// Description section

// <code2> <code1> <code2> <code1> <code1>
3ab4-12-AB-34;
#Xaeiou-1XYZZZ ;
12-ABCD:CD:4321-12-ab;
#abcd118 ;
#aefghi12IJK;

$$$$$$$
// Simulation section

// First part: OXYGEN and CELLS instructions (oxygen=10, cells=4)
OXYGEN 10;
CELLS 4;

// Second part: MOD_STATE1 and MOD_STATE2 commands

// The quantity of oxygen decreases of 3 units (oxygen=7, cells=4)
MOD_STATE1 OXYGEN - MAX(1,3,2);

// The quantity of cells increases of 2 units (oxygen=7, cells=6)
// MAX(1,MAX(1,2),MAX(0,1),1) = MAX(1,2,1,1) = 2
MOD_STATE1 CELLS + MAX(1,MAX(1,2),MAX(0,1),1);

// + 3 * 1.1 + 5 * 0.9 - 2 * 1.1 = + 5.6 = 5
// The quantity of oxygen increases of 5 units (oxygen=12, cells=6)
MOD_STATE2 TEMP 1.1 FOOD 0.9 OXIGEN : + 3 TEMP, + 5 FOOD, - 2 TEMP;
```

Output:

```
oxygen=7 cells=4
oxigen=7 cells=6
oxigen=12 cells=6
```