Formal Languages and Compilers

05 July 2017

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 *car* sections, separated by means of the two characters "$$". A "%" character identifies the start of a comment, which is defined from the "%" character to the end of the line.

Header section: lexicon

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

- <token1>: an even number between −124 and 86, optionally followed by a word of at least 5 lowercase alphabetic letters (in an odd number), followed by the word "ABC" or by 3 or more repetitions of the words ("XX", "XY", "YX" or "YY"), which can appear in any possible combination (e.g., XXYXYX, XXXXYYYY, XYXYXY, ...).
- <token2>: it is composed of an odd number of words, at least 5, separated by the character "*" or "-". A word is a binary number between 10 and 11110.
- <token3>: a hour with the format "HH:MM:SS" between 08:12:34 and 17:21:37.

Header section: grammar

In the header section <token1> and <token2> can appear in **any order** and **number** (also **0 times**), instead, <token3> can appear only **0, 1, 2 or 3 times**.

Car section: grammar and semantic

The *car section* starts with the `set` instruction, which sets the state of a car in terms of position (coordinates X and Y) and fuel (F). The `set` instruction has the syntax `set position X, Y - fuel F;`, where `X`, `Y` and `F` are *signed integer* numbers, `position X, Y` sets the position of the car to the values `X` and `Y`, while `fuel F` sets the fuel to `F`. The **order** of `position X, Y` and `fuel F` inside the `set` instruction can be **inverted**, and both parts can be **optional**. In the case they are not present, the `X`, `Y` variables (or `F`) are set to 0. Examples: `set - position 1, 2;` (sets X: 1 Y: 2 F: 0), `set - ;` (sets X: 0 Y: 0 F: 0).

After the `set` instruction, there is a list of <commands>. The number of commands is **even** and **at least 4**. The following three commands are defined:

- `declare`: This command is the `declare` word, followed by a list of <attributes> between braces (separated with commas) and followed by a <variable_name>. Each <attribute> is an <attribute_name> (a string of letters and numbers starting with a letter), an "=", a <signed_integer> number and a semicolon. This command stores the tuples <attribute_name>, <signed_integer> into an hash table with key <variable_name>. **The hash table is the only global variable allowed in all the examination, and it must only contain the information derived from the `declare` command. Solutions using other global variables will not be accepted.**
- `if`: This command stats with the character `?`, followed by a <boolean_exp> and by a list of `mv` instructions (<$\texttt{mv\_list}_1$>) between braces. Optionally, it is followed by an `else` branch with the following grammar `else` { <$\texttt{mv\_list}_2$> }. If the evaluation of <boolean_exp> is TRUE, the `mv` instructions inside <$\texttt{mv\_list}_1$> are executed. Otherwise, if the evaluation of <boolean_exp> is FALSE and the `else` branch exists, the `mv` instructions inside <mv_list_2> are executed. <boolean_expr> can manage only the following operators: comparison (`==`), logical (`and`, `or`, `not`), and round brackets. The comparison operator can be applied only between a <variable_name_and_attribute> and a signed integer number. <variable_name_and_attribute> is a <variable_name>, followed by a "." and by an <attribute_name>. The value associated to a <variable_name_and_attribute> can be accessed through the global hash table.

- `fuel`: The `fuel` command *increases* or *decreases* the fuel of the quantity computed by the *min* or *max* functions. This command is composed by the word `fuel`, followed by the word `increases` or `decreases` (which specifies an increasing or a decreasing on the fuel quantity, respectively), a ":" character, and the *min* or *max* function. The *min* function is the word `min`, followed by a "(", by a <list_of_values> and by a ")". <list_of_values> is a list of <variable_name_and_attribute> or signed integer numbers separated with commas. The function *min* computes the minimum between the elements listed in <list_of_values>. The function *max* has a syntax similar to the function *min*, and it computes the maximum between the values listed in <list_of_values>. See the example.

All commands are ended with the ";" character. Usage examples of these commands are reported in the *Example* section. The `mv` instructions are used only in the `if` command. A `mv` instruction, with the grammar `mv X, Y, fuel F;`, modifies the position and the fuel of the car of the quantities X, Y and F, respectively. Values X, Y and F are signed integer numbers. In addition, the `mv` instruction prints the new state (position and fuel) of the car. **The state of the car can not be stored into a global variable. Use instead inherited attributes to access from the parser stack the old state, and save the new state in the parser stack.**

Goals

The translator must execute the language previously described.

Example

Input:

```
-14abcdefgABC;                % <token1>
10*100-101-10*1000-11110-10; % <token2>
08:12:34;                     % <token3>
16:01:59;                     % <token3>
82YXYXXXYX;                   % <token1>

$$

set position 2, 3 - fuel 10;       % or other possible implementations of set

declare {x=3; y=7; tires=5;} car;  % stores in hash table: car.x=3, car.y=7, car.tires=5

? car.x==3 and car.y==5 or not car.tires==6 {       % TRUE and FALSE OR NOT FALSE = TRUE
  mv -1, -1, fuel -1;                              % prints new state: X: 1 Y: 2 FUEL: 9
  mv -1, -2, fuel -2;                              % prints new state: X: 0 Y: 0 FUEL: 7
};

fuel increases : max ( 3, car.x, car.y );   % The maximum is 7 (i.e., car.y)
                                            % new state is: X: 0 Y: 0 FUEL: 14

? car.y==4 {
  mv -1, -1, fuel -1;
}else{
  mv -2, -2, fuel -3;    % prints new state: X: -2 Y: -2 FUEL: 11
  mv -1, -2, fuel -2;    % prints new state: X: -3 Y: -4 FUEL: 9
};
```

Output:

```
X: 1 Y: 2 FUEL: 9
X: 0 Y: 0 FUEL: 7
X: -2 Y: -2 FUEL: 11
X: -3 Y: -4 FUEL: 9
```