Formal Languages and Compilers

19 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 *state* sections, separated by means of three or more "#" characters. In the input file C-style comments are allowed (i.e., `/* comment */`).

Header section: lexicon

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

- <token1>: it is the character "?" followed by a binary number containing 2 or 4 characters "1" (e.g., ?0100010, ?011010001), or it is the character "?" followed by a word composed by the symbols "x" and "y" without consecutive equal symbols (i.e., near to the symbol "x" we can have only "y", near the symbol "y" we can have only "x"). Example: ?xyxy, ?xyxyxyx, ?x, ?.
- <token2>: it is a date with the format "YYYY/MM/DD" between 2017/01/18 and 2017/07/02. Remember that the month of February has 28 days and the months of April and June have 30 days. The date is optionally followed by a hour with the format ":HH:MM" between :01:12 and :11:37.
- <token3>: it is composed by at least 6 words in even number (i.e., 6, 8, 10,...). Each word is an odd number between 15 and 1573. Words are separated by means of the characters "/", "$" or "+".

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 <token1>, followed by **2** or **3** or **9** repetitions of <token2> and <tolen3>, which can appear in **any possible order**.

   Example: <token1> <token1> <token1> <token1> <token2> <tolen3> <token2>

2. <token1> and <token2> can appear in **any order** and **number** (**also 0** times), instead, <token3> must appear **exactly 2 times**. The **first token** of the sequence **must** be <token3>.

   Example: <token3> <token1> <token2> <token3> <token2> <token1> <token2>

State section: grammar and semantic

The *state section* starts with the `CONFIGURE` instruction, which sets the *humidity* of a room to `H` and its *temperature* to `T`. The `CONFIGURE` instruction has the syntax `CONFIGURE HUMIDITY H TEMPERATURE T;` or `CONFIGURE TEMPERATURE T HUMIDITY H;`, where `T` and `H` are *signed integer* numbers. At **most one** of the two quantities (`TEMPERATURE T` or `HUMIDITY H`) can be **omitted**, in this case the default *humidity* is `H=50`, and the default *temperature* is `T=20`. Example: `CONFIGURE HUMIDITY 75;` (sets `H=75` and `T=20`).

After the `CONFIGURE` instruction, there is a list of <commands>, possibly **empty**.

The following two commands are defined:

- `STORE`: This command is the `STORE` word, followed by a list of <assignments> (separated with commas). Each <assignment> is a <variable_name> (a letter or an "_" character, followed by letters, numbers or "_" characters), an "=" and an <arithmetic_expression>. Such elements can be stored into a *data structure* (hash table, list,...). **This data structure is the only global data allowed in all the examination, and it must only contain the information derived from the `STORE` command. Solutions using other global variables will not be accepted.** An <arithmetic_expression> is a typical arithmetic expression containing +, -, *, / and ^ (power) as operators; signed integer numbers, <variable_name> (which value can be accessed through the *data structure*) or the *avg* function as operands, and round brackets. The *avg* function has the syntax `avg(<arithmetic_expression_list>)` or `AVG(<arithmetic_expression_list>)`, where each element of the list is an `<arithmetic_expression>` (elements are separated with commas). This function computes and returns the average value, truncated to a signed integer, between the listed elements.

<arithmetic_expression_list> can be **empty**, in this case the function returns the value 0 (e.g., `AVG()` returns 0). Examples of <arithmetic_expression>: 3*(2+a); 3+a*(avg(2,3,AVG(2,a),a+3)+2^ 3).

- `CASE`: The `CASE` command has the following syntax:

  `CASE` <arithmetic_expression*> `IS { <conditions_list> } ;`

  Two types of <conditions> are defined:

  1. `IN RANGE A, B { <mod_list> }`
  2. `EQUAL C { <mod_list> }`

  where `A`, `B` and `C` are <arithmetic_expression>. For the `IN RANGE` condition, if the result of <arithmetic_expression*> is in the range $[A...B]$, the *mod* instructions listed in <mod_list> are executed. Conversely, for the `EQUAL` condition, the *mod* instructions listed in <mod_list> are executed only when <arithmetic_expression*> is equal to `C`. For any `CASE` command, at most one condition can be TRUE. The *mod* instructions (with grammar `TEMPERATURE T;` or `HUMIDITY H;`) modify the current value of the room *temperature* or *humidity* of the quantities `T` or `H`, which have to be summed to the current tempetature or humidity state values, respectivelly. In addition, *mod* instructions display the new state (temperature and humidity) of the room. <mod_list> can be **empty**. **The state of the room, temperature and humidity, cannot 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.**

All commands are ended with the ";" character. Usage examples of all the commands are reported in the *Example* section.

Goals

The translator must execute the language previously described.

Example

Input:

```
15$17$1501+245/19/255/1503/145; /* <token3> */
?000100101001;                  /* <token1> */
2017/07/02:01:40;               /* <token2> */
?xyx;                           /* <token1> */
1573$1573+15+15+15+15;          /* <token3> */
2017/03/30;                     /* <token2> */

####

CONFIGURE TEMPERATURE 20 HUMIDITY 60;

STORE a = 3+2, b = a+2, c = 0+avg(2);     /* a=5; b=7; c=2; */

CASE a IS { /* a=5 */
  IN RANGE b-5, 4 {     /* FALSE, because a=5 is not in the range [2,4] */
    HUMIDITY -3;
  }
  EQUAL b-2 {           /* TRUE, because a=5 is equal to b-2=7-2=5 */
    TEMPERATURE -3;                     /*                   Prints new state: T: 17 H: 60 */
    HUMIDITY 3+a-b;                     /* humidity 3+5-7=1. Prints new state: T: 17 H: 61 */
    TEMPERATURE AVG(2, 2, avg(c, c, 2)); /* avg(2, 2, 2)=2.   Prints new state: T: 19 H: 61 */
  }
};

CASE a+avg(0,2) IS { /* a+avg(0,2)=5+1=6 */
  EQUAL 2 { }
  EQUAL 6 {      /* TRUE because a+avg(0,2) = 6 */
    TEMPERATURE 2+avg();  /* 2+avg()=2+0=2. Prints new state: T: 21 H: 61 */
    HUMIDITY 3+2*3;       /* 3+2*3=3+6=9.   Prints new state: T: 21 H: 70 */
  }
  IN RANGE 2, 5 { temperature +2; humidity -3; }
};
```

Output:

```
T: 17 H: 60
T: 17 H: 61
T: 19 H: 61
T: 21 H: 61
T: 21 H: 70
```