Formal Languages and Compilers

3 September 2015

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

Input language

The input file is composed of three sections: *header*, *cars* and *race* sections, separated by means of an **even number** of "#" characters (**at least** 4). C++ stile comments (i.e., `// <comment>`) can be present in the input file. ***("####")("##")****

Header section: lexicon ***>> NO SEMANTICS >> testing regex abilities + token acquirement***

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

these 3 tokens mostly contribute to regex total points

- <token1>: it starts with the character "%" repeated an odd number of times, at least 5, or it starts with 2 o 3 repetitions of the words "**" and "???" in any combination (`???**`, `??????`, `**??????`, `******`,...). The first part of the token is optionally followed by an odd number between −35 and 333.
- <token2>: it is composed of two dates separated by the character "-" or by the character "+". A date has the format YYYY/MM/DD and it is in the range between 2015/12/12 and 2016/03/13, with the exclusion of the day 2016/01/05. Remember that the month of February has 29 days.
- <token3>: it is the character "$" followed by a binary number between 101 and 101000

Header section: grammar

In the *header* section, <token2> and <token3> must appear **exactly 1 time**, instead, <token1> can appear **0 or more times**. Tokens can appear in **any order**.

Cars section: grammar and semantic

The *car section* is composed of a list of <cars>. The number of <cars> is **even** and **at least 2**. Each element of the list is composed of a <car_name> (a quoted string), followed by a "{", a non-empty list of <speeds> in which elements are separated by commas, and a "}".

A <speed> is composed of a <section_name> of the race track (a quoted string), an "=", a <speed_value> (an unsigned integer number) and the word "m/s".

At the end of this section, the translator must have filled a hash table that contains all the information regarding cars, and with keys the <car_name>. **The hash table is the only global variable allowed in all the examination, and it must only contain the information reported in the *car section* (i.e., after this section its content cannot be written, but only read). Solutions using other global variables will not be accepted.**

Race section: grammar and semantic

The *race section* is composed of a list, **eventually empty**, of <print_min_max> functions followed by a **non-empty** list of <performances>.

Each <print_min_max> function is composed of the word "PRINT_MIN_MAX", followed by a "(", by a <car_name>, a ")", a "(", a **non-empty** list of <section_names>, a ")" and a ";". This function must print the *minimum* and the *maximum* values of the <speeds> of the <section_names> listed in

the second part of the function and referred to the <car_name> reported in the first part of the function. The <speed> values can be obtained from the hash table.

Each <performance> (i.e., an element of the list of <performances>) is composed of a <car_name>, the symbol "->", a list of <parts> (which elements are separated by the character "|"), and a ";". Each <part> is composed of a <part_name> (i.e., the word "PART" followed by an unsigned integer number), a ":" and a list of <drive_stats> in which elements are separated by commas. Each element of the <drive_stats> list is composed of a <section_name>, a <distance> (i.e., an unsigned integer number) and the character "m".

The translator must compute, for each part, the time needed for the specific <car_name> to finish the part. To compute this time, the <distance> must be divided by the <speed>, obtained by using <section_name> as a key in the hash table. **To this extent <car_name> must be accessed through inherited attributes.**

At the end of each <performance>, the total time to finish all the parts must be printed.

At the end of the list of <performances>, the <car_name> that finished all the parts with the small amount of total time and the total time must be printed. **Produce exactly the output as reported in the example.**

Goals

The translator must execute the language of the *cars* and *race* sections.

Example

Input:

this part usually doesn't contain any semantic >> only for checking regex usage + tokens

```
// Header section
%%%%%-31;                 // <token1>
$10111;                   // <token3>
???**???45;               // <token1>
2015/12/25+2016/02/28; // <token2>
####
// Cars section

"CAR_A" {
  "curve_1" = 10 m/s,
  "straight_1" = 50 m/s,
  "curve_2" = 20 m/s,
  "straight_2" = 60 m/s,
  "curve_3" = 15 m/s,
  "straight_3" = 80 m/s
}
"CAR_B" {
  "curve_1" = 5 m/s,
  "straight_1" = 80 m/s,
  "curve_2" = 20 m/s,
  "straight_2" = 75 m/s
  "curve_3" = 10 m/s,
  "straight_3" = 80 m/s
}
######
// Race section
// between 10, 50 and 15 the min is 10 and the max is 50
PRINT_MIN_MAX("CAR_A")("curve_1", "straight_1", "curve_3");

"CAR_A" ->
  PART1 : "curve_1" 20 m |  // 20/10=2 s
  // 400/50+40/20+540/60=8+2+9=19 s
  PART2 : "straight_1" 400 m, "curve_2" 40 m, "straight_2" 540 m |
  PART3 : "curve_3" 30 m, "straight_3" 640 m  // 30/15+640/80=2+8=10 s
;
"CAR_B" ->
  PART1 : "curve_1" 20 m |  // 20/5=4 s
  // 400/80+40/20+540/75=5+2+7.2=14.2 s
  PART2 : "straight_1" 400 m, "curve_2" 40 m, "straight_2" 540 m |
  PART3 : "curve_3" 30 m, "straight_3" 640 m // 30/10+640/80=3+8=11 s
;
```

in this section i create a symbol table containing two symbol tables for car A and car B, each containing all possible values for each curve / straight

i then proceed to the second part and apply all necessary calculations

here i need to use the parse tree to calculate the given function

Output:

```
MIN: 10 MAX: 50
"CAR_A"
PART1: 2.0 s
PART2: 19.0 s
PART3: 10.0 s
TOTAL: 31.0 s
"CAR_B"
PART1: 4.0 s
PART2: 14.2 s
PART3: 11.0 s
TOTAL: 29.2 s
WINNER: "CAR_B" 29.2 s
```