Formal Languages and Compilers

12 July 2016

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 *program* sections, separated by means of the two characters "##". C stile comments (i.e., `/* <comment> */`) are allowed in the input file.

Header section: lexicon

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

- <token1>: it begins with the character "?", followed by an even number (at least 4) of lowercase alphabetic letters or by an odd number (at least 5) of uppercase alphabetic letters. The first part of the token is optionally followed by a binary number between 110 and 10101.
- <token2>: it is composed of an even number of words, separated by the character "+" or "-". Each word is composed of 3 or 6 repetitions of the strings "ij", "ji", "ii" and "jj" in any combination.
- <date>: a date has the format YYYY/MM/DD and it is in the range between 2016/05/28 and 2016/08/15, with the exclusion of the day 2016/06/24. Remember that the month of June has only 30 days.

Header section: grammar

Each token can appear in the header **in any order and number** (**also 0**), with the only exception of <date>, that can appear only **zero**, **one** or **two** times. Remember to manage this requirement with grammar.

Program section: grammar and semantic

The *program section* is a list of <instructions>. The number of <instructions> is **odd** and **at least 5**. Two types of instructions are allowed: <assign> and <switch>. They can appear in the *program section* in **any order**, and both are terminated by a ";".

The <assign> instruction is a <var> (i.e., a letter, followed by letters and digits), an "=" and an <exp> (expression). After the execution of an <assign>, the result of <exp> has to be stored into an entry of a hash table with key <var>. **The hash table is the only global variable allowed in all the examination, and it can contain only results of <assign> instructions**. Each time an entry is inserted into the hash table, the translator has to print it on standard output. <exp> is a typical arithmetical expression (which includes unsigned integer numbers or <var>, parenthesis, and "+", "-", "*" and "/" operators), e.g., $2 + (3 * a)$. As operands, <exp> can also contain two functions:

- `stat(<operation>,` exp_1`,` exp_2`,...,`exp_n`)`: `<operation>` can be `MIN` or `MAX`. If `<operation>` is equal to `MIN`, the function `stat` returns the minimum value between the results of expressions $exp_1, exp_2, ..., exp_n$. If `<operation>` is equal to `MAX`, this function returns the maximum. See the example. The list of expressions ($exp_1, exp_2, ..., exp_n$) can be **empty**, in this case the `stat` function returns the value 0 (e.g., `stat(MIN)` returns 0).

- `case(<var>,` $exp_1^{if}:exp_1^{res}$ `,` $exp_2^{if}:exp_2^{res}$ `, ... ,` $exp_n^{if}:exp_n^{res}$`)`: The command `case` is composed of a list of actions (exp_1^{if}:exp_1^{res} , exp_2^{if}:exp_2^{res} , ... , exp_n^{if}:exp_n^{res}) separated by commas. The command `case` executes a specific action i of the list (exp_i^{if}:exp_i^{res}) if the result of the expression exp_i^{if} is equal to the value stored in the hash table for the entry with key <var>. In this case, when the equality between exp_i^{if} and the value of <var> is verified, the function `case` returns the result of expression exp_i^{res}. At most one action between those listed inside the `case` command is verified (i.e., exp_i^{if} is equal to the current value of variable <var> for only one element of the list of actions exp_1^{if}:exp_1^{res} , exp_2^{if}:exp_2^{res} , ... , exp_n^{if}:exp_n^{res}). If no condition is verified, the function `case` returns the value 0. See the example.

The second type of instruction is <switch>. It is the word "switch" followed by a <var>, a "{" a <list_of_cases>, and a "}". <list_of_cases> is a **non-empty** list of <case>. Each <case> has the grammar: `case <exp>: <list_of_prints>`. If <exp> is equal to the content of <var>, the <print> commands of <list_of_prints> are executed. In particular, each <print> has the grammar: `print` `<`exp^{print}`>;`.

Each <print> command has to access with inherited attributes to the values <var> and <exp> and, if the content of <var> is equal to <exp>, it has to print the value <exp> multiplied by < exp^{print} >. More than one `case` can be verified (i.e., its <exp> is equal to the content of <var>) inside the same <switch> instruction.

For `case` function and <switch> instruction use inherited attributes.

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 */
ijjiii-iijjii+ijijij+iijjijjiiijj;  /* <token2> */
?HELLO1001 ;                        /* <token1> */
2016/05/28;                         /* <date>   */
?helloworld;                        /* <token1> */
2016/06/20 ;                        /* <date>   */
##
/* Program section */
a = 5;                                         /* Assignment: a=5 */
b = ( 3-1 ) * 1;                               /* b=2*1=2 */
c = stat(MIN, b, a-1, 2+3);                    /* Minimum between 2 (b), 4 (a-1) and 5 is 2 -> c=2 */
d = case(b, 3:a+2, 2 : b+2 , 6:0);             /* Since b is equal to 2, value assigned to d is d=b+2=2+2=4 */

e = case(a, stat(MAX, 2, stat(MAX,5,6)) : 3,         /* stat(MAX, 2, 6) = 6 -> action 6:3 */
            5 : case(b, 3:stat(MIN,3,2), 2:4, 7+8:6) /* case(b, 3:2, 2:4, 15:6)=4 (because b=2) -> action is 5:4 */
    );     /* The second action of the outer case (5:4) is executed (because a=5). At the end e=4 */

f = case(a, 1:0, 2:3) + stat(MIN, 2, 3);       /* f=0+2=2 */

switch a {
  case 1+1: print 3;              /* if a is equal to 2 (FALSE in this case) -> printed value is 6 (3*2) */
  case 5: print 3;                /* it prints 15 (5*3) because a is equal to 5 */
          print stat(MIN, 3, 2);  /* it prints 10 (5*2) because a is equal to 5 */
  case 3: print 2+5;              /* if a is equal to 3 (FALSE in this case) -> printed value is 21 (3*7) */
};
```

Output:

```
a=5 b=2 c=2 d=4 e=4 f=2 15 10
```