Formal Languages and Compilers

20 July 2020

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 *commands* sections, separated by means of the two characters "$$". Comments are possible, and they are delimited by the starting sequence "(++" and by the ending sequence "++)".

Header section: lexicon

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

- `<token1>`: it starts with a word composed of at least 7 characters in the set "a" , "b" or "c", disposed in any order and, in total, in odd number (e.g., abcabca, aabbccabc). The word is followed by the character "#", and optionally by a hexadecimal and even number between −5C and aB6. Remember that even hexadecimal numbers are those ending with 0, 2, 4, 6, 8, A or a, C or c, and E or e .
- `<token2>`: it is a hour with the format "HH:MM:SS" between 07:13:24 and 17:37:43, followed by the character ":" and by a binary number between 101 and 11010.

Header section: grammar

The header section contains one of these two possible sequences of tokens:

1. at **least 5**, and in **odd** number (5, 7, 9,...) repetitions of `<token1>`, followed by **3 or 21** repetitions of `<token2>`
2. **three** `<token2>` and **any number** of `<token1>` (**even 0**). This sequence **must start** with a `<token2>`, the second and third repetitions of `<token2>` can be in **any position** of the sequence.

Commands Section: grammar and semantic

The *commands section* is composed of a list of `<commands>`, which can be possibly **empty**.

Two types of commands are possible:

- *Assignment*: it is a `<variable>` (same regular expression of C identifiers), followed by a "=", by an `<expr>`, and by a `;`. This command stores the result of `<expr>` into an entry of a global symbol table with key `<variable>`. **This symbol table is the only global data structure allowed in all the examination, and it can be written only by means of an assignment command**.
- *Compare*: it has the following syntax:

  `compare <expr`$_1$`> with <comp_list> end ;`

  where `<comp_list>` is a non-empty list of `<comp>`, where each `<comp>` has the following syntax:

  `<expr`$_2$`> { <print_list> }` . For each `<comp>` for which the result of `<expr`$_2$`>` is equal to the result of `<expr`$_1$`>` (more than one `<comp>` can meet this requirement within the same `compare` command), all the `<print>` commands listed in `<print_list>` are executed. A `<print>` command is the word `print` followed by an `<expr>` and followed by a `;`. The `<print>` command prints the results of `<expr>` into the screen. **For the implementation of the `compare` instruction, within the grammar rule of the `<print>` command, use inherited attributes to access the values of `<expr`$_1$`>` and `<expr`$_2$`>`, and to decide to execute or not the print action.**

An `<expr>` is a typical arithmetical expression, which includes integer numbers or `<variable>`, parenthesis, and “+”, “-”, “*” and “/” operators. An example of `<expr>` is $3 + (6 * a)$.

Goals

The translator must execute the language, and it must produce the output reported in the example. For any detail not specified in the text, follow the example.

Example

Input:

```
(++ Header section (second type of grammar for the header) ++)
07:13:24:101;       (++ <token2> ++)
08:13:10:11001 ;    (++ <token2> ++)
aabbccabc#-a;       (++ <token1> ++)
10:13:26:1000;      (++ <token2> ++)

$$
```

"(++" ~"++)" >> comment

```
(++ Commands section ++)
a = 2;                (++ assigns 2 to var a ++)
b = 2 + 3 * 2;        (++ assigns 2+6=8 to var b ++)
c = (a + 1) * 2;      (++ assigns 3*2=6 to var c ++)
```

create symbol table and insert declarations (a,b,c)
remember to use precedence rules (lab 4)\

```
compare 3+a with      (++ 3+a=5 ++)
  3*a {               (++ FALSE ++)
    print 3;
    print a+1;
  }
  1+ 2*2 {            (++ TRUE ++)
    print 2;          (++ print 2 ++)
    print 3;          (++ print 3 ++)
  }
  b-3     {           (++ TRUE ++)
    print a*2;        (++ print 4 ++)
  }
end;
```

these are not executed

these are executed

How ? >

here we have
listPrint ::= PRINT E:X SC | listPrint PRINT E:x SC
{: if(stack(*1) == stack(*2) {sys.out.println(x)}:}

this is executed

to access the value to compare with (in this case "3+a" we can either use a marker, and place it after the *compare_list NT0 compare_item)* or access directly the value, which would in this case be parser.stack(-x) where x depends
on how we built the grammar

Output:

```
2
3
4
```

Weights: Scanner 8/30; **Grammar** 9/30; **Semantic** 10/30