Formal Languages and Compilers

21 October 2022

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 *command* sections, separated by means of the sequence of 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 3 types of tokens, each terminated with the character ";":

- `<tk1>`: it begins with the character "?", followed by a word composed of an even number (at least 6) of uppercase alphabetic letters. It is then followed by an octal number (possible digits are from 0 to 7) between −127 and 323 , and optionally followed by 4 or more repetitions of the words "xx", "yy", or "zz" in any combination.
- `<tk2>`: it is composed by 2, 12, or 15 emails, where each email is a word composed of numbers, letters and characters "_" and ".", the character "@", and a word composed of letters and numbers, a ".", and the word "it", "org", or "com". Each email is separated by the character "!" or "/".
- `<tk3>`: it is the word "tk3".

Header section: grammar

In the *header section* the tokens `<tk1>` and `<tk3>` can appear in **any order and number** (**even 0** times), instead, `<tk2>` can appear only **0, 1, or 4 times**.

Code section: grammar and semantic

The *command section* is composed of a list of `<commands>`. The list can be possibly **empty**, or with an **even** number of elements, **at least 4**. As a consequence, the list can be composed of 0, 4, 6, 8,... elements.

Two types of commands are possible:

- *Assignment*: it is a `<variable>` (same regular expression of C identifiers), followed by a "=", and a `<bool_expr>`. This command stores the result of the `<bool_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**. Each time an *assignment* command is executed, the command prints into the screen the `<variable>` name and the associated value.
- *CMP*: it has the following syntax:

  `CMP <bool_expr> <actions_list>`

  where `<bool_expr>` represents the result of a boolean expression (i.e., a `T` (true) or a `F` (false) values). `<actions_list>` is a list of at least one `<action>`, where an `<action>` is the word `WITH`, a `<bool_expr_a>` (same regular expression of `<bool_expr>`), the character "`[`", a `<print>` instruction, and the character "`]`". The `<print>` instruction is the word `print`, followed by a "(", a *quoted* *string*, a ")", and a ";". The `<print>` instruction is executed each time the result of `<bool_expr>` equals the result of `<bool_expr_a>`.

`<bool_expr>` can contain the following logical operators: `AND`, `OR`, `NOT`, and round brackets. Operands can be `T` (the true constant), `F` (the false constant), a `<variable>` (which represents the value stored in the symbol table by an *assignment* command), and the `fz_and()` function. The `fz_and()` function takes in input a list of `<bool_expr>` separated by a ",", and returns the "logical and" of the results of the listed `<bool_expr>` (i.e., the `fz_and()` function returns `T` only if the results of all the listed `<bool_expr>` are `T`, otherwise it returns `F`).

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:

```
tk3 ;                                            {++ tk3 ++}
name1.surname1@skenz.it/name2.surname2@abc.net; {++ tk2 ++}
?ABCFEF-36xxyyxxyy ;                             {++ tk1 ++}
tk3;                                             {++ tk3 ++}

$$$ {++ division between header and command sections ++}

x1 = T;
x2 = NOT T AND NOT x1 ;  {++ F AND F = F ++}

{++ fz_and(T, T, T, F) OR F = F OR F = F ++}
x3 = fz_and(T, T, fz_and(T, x1), F  ) OR F;


CMP T AND F   {++ T AND F = F ++}
WITH F OR F [   {++ executed ++}
  print("one");
]
WITH x1 [      {++ not executed ++}
  print("two");
]
WITH NOT x1 [  {++ executed ++}
  print("three");
]
```

Output:

```
x1 T
x2 F
x3 F
"one"
"three"
```

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