Formal Languages and Compilers

22 July 2015

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

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> */`) can be present in the input file.

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

- <token1>: a hour with the format "HH:MM" (between 11:35 and 13:51) or in the 12 hours format with the words "am" or "pm" to indicate the morning or the afternoon (between the same range, i.e., between 11:35am and 01:51pm)
- <token2>: is a string between double quotes ("`$ abc ?!`"), followed by the character ":", followed by an IP address (ex. "`abc &+`"`:130.192.18.123`)).
- <token3>: starts with the character "%", followed by an even number of alphabetic characters (at least 4), followed by an odd number between -35 and 253, and optionally ended with 4 or more characters "$*$", or with the word YXY (where the number of X must be odd: YXY, YXXXY, YXXXXXY,...).

Header section: grammar

In the *header* section, <token1> and <token2> can appear in **any order and number** (also **0 times**). Instead, <token3> must appear **exactly 2 times**.

Program section: grammar and semantic

The *program section* is composed of a list of <instructions> (use a **right recursive** list). The number of <instructions> is **even** and **at least 4**. The programming language defines 3 types of instructions: `EVALUATE`, `CASE` e `SAVE`. All the three instructions make use of <boolean_expressions>.

To simplify the programming language, <boolean_expressions> can contain only the following operators: comparison (`==`), logical (`&&` (and), `||` (or), `!` (not)), arithmetic (`+`, `*`) and round brackets. The arithmetic and comparison operators can be applied to integer numbers, the logical operators can be used with boolean values (i.e., tokens TRUE and FALSE, or the result of a comparison operation) or with <variables> (a <variable> is a word that begins with a letter or the character "_" and followed by letters, numbers and characters "_"). The boolean value of a specific <variable> can be accessed through a global hash table. Three examples of <boolean_expressions> are: `3*(5+2) == 3 && !a || b`; `a && (b || TRUE) || 4 == 3`; `TRUE AND FALSE`. Each <instruction> is terminated with the character ";".

- **EVALUATE <boolean_expression>;** executes <boolean_expression> and print the result (i.e., `TRUE` or `FALSE`).
- **CASE_TRUE <boolean_expression1>, CASE_FALSE <boolean_expression2>;** makes use of the results obtained in the previous instruction (I_{-1}). If I_{-1} is equal to `TRUE` the expression <boolean_expression1> is executed, otherwise the expression <boolean_expression2> is executed. This instruction prints the result of the executed boolean expression.

- **SAVE <assignment_list>;** allows to store boolean values in a global hash table (**this hash table is the only global variable allowed in all the examination**). <assignment_list> is a **non empty** list of <assignments>. An <assignment> is a <variable> followed by the character "=" and followed by a <boolean_expression> or an <offset>. <variable> is a *key* of an *entry* of the hash table, the result of a <boolean_expression> or the boolean value related to a specific <offset> is the *value* associated with the *key*.

  An <offset> is the letter "o" followed by a negative integer number between square brackets (i.e., o[-5], o[-6],...). In particular, $o[-1]$ represents the result of the instruction I_{-1} (i.e., the last result), $o[-2]$ is the result of the instruction I_{-2} (i.e., the result of the last but one instruction), an so on. For instance, the instruction `SAVE x=o[-3], y=TRUE && b;` stores the result of the instruction I_{-3} (i.e., last but two) in the variable `x`, and the result of the boolean expression `TRUE && b` in the variable `y` (`b` is a <variable>, whose value can be accessed through the hash table). The instruction SAVE provides and prints always the result FALSE.

Goals

The translator must execute the language of the *program* section. Can be assumed that the input file is correct (for instance the `CASE` instruction cannot be the first one, because in this case I_{-1} is not defined).

Advice: Define the list of <instructions> as a **right recursive** list, in order to leave the results of the previous instructions into the parser stack (the results of the previous instructions can be accessed using inherited attributes).

Example

Input:

```
"hello £$%&":130.192.12.17 ; /* <token2> */
%abcdef-1YXXXY ;             /* <token3> */
11:37;                       /* <token1> */
01:50pm ;                    /* <token1> */
%xyzj3;                      /* <token3> */

##

/* TRUE && FALSE || TRUE = TRUE */
EVALUATE TRUE && (1+2)*3 == 4 || TRUE;

/* x=TRUE */
SAVE x=o[-1]; /* FALSE */

EVALUATE 2 == 3; /* FALSE */

/* CASE_FALSE: FALSE && FALSE = FALSE */
CASE_TRUE x, CASE_FALSE FALSE && 8+3 == 9;

/* i=FALSE; j=FALSE; k=TRUE */
SAVE i=o[-1], j = TRUE && FALSE, k=o[-4]; /* FALSE */

/* ! FALSE && TRUE = TRUE AND TRUE = TRUE */
EVALUATE ! i && 2 == 2;
```

Output:

(The values of the variables stored in the hash table are `x=TRUE`, `i=FALSE`, `j=FALSE` e `k=TRUE`)

```
TRUE
FALSE
FALSE
FALSE
FALSE
TRUE
```