

Formal Languages and Compilers

21 September 2017

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 three sections: *header*, *declaration* and *state* sections, separated by means of 4 or more “*” characters. In the input file, C++-style comments are allowed (i.e., `// comment`).

Header section: lexicon

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

- `<token1>`: it is composed by at least 3 dates in odd number (i.e., 3, 5, 7,... dates). Each date has the format “YYYY/MM/DD” and it ranges between 2017/07/02 and 2017/10/21 (remember that the month of September has only 30 days). Dates are separated by means of the characters “-” or “#”.
- `<token2>`: it is the character “\$”, followed by a word containing 2 or 5 vowels (i.e., a, e, i, o, u). The other letters of the word are lowercase alphabetic consonants (i.e., letters in the range b...z, with the exclusion of vowels). Examples: \$hello or \$education. Alternatively, the “\$” character can be followed by a binary number between 10 and 101001.
- `<token3>`: it starts with a word composed of at least 4 character in the set “@”, “%” or “&”, disposed in any order and in even number (e.g., %@@&, @%@@%%). This first part of the token is optionally followed by an odd number between -43 and 1231.

Header section: grammar

In the header section `<token1>` must appear exactly **1 time**, `<token2>` must appear exactly **2 times**, while `<token3>` can appear **0 or more times**, and in **any** position. Manage this requirement with the grammar.

Declaration section: grammar and semantic

It is composed of a list that, if it is not **empty**, it contains **at least 3 <declarations>**. The number of `<declarations>` is **odd**. Each `<declaration>` is composed by an `<id_name>`, the symbol “=”, an `<id_descr>`, a “{”, a list of `<attributes>`, a “}” and a “;”. Each `<attribute>` is an `<id_attr>`, a “=”, a `<value_attr>` (an unsigned integer number) and a “;”. `<id_name>`, `<id_descr>` and `<id_attr>` have the same regular expression of identifiers in the C programming language (i.e., the first character is a letter or an “_”, possibly followed by letters, numbers and “_” in any combination). All the semantic information defined in this section and useful in the following section, can be stored into a global structure. **This global structure is the only global data allowed in all the examination, and it must only contain the information derived from this section and from the *Assignment* command described in the following. Solutions using other global variables will not be accepted.** For each `<declaration>`, the translator has to print the minimum and the maximum values between the `<value_attr>` listed in the declaration.

State section: grammar and semantic

The *state section* starts with the **START** instruction. It is the words **START STATE**, followed by an `<id_state>` (a C identifier) and terminated with a “;”. This instruction sets the *current state* to `<id_state>`.

After the **START** instruction, there is a list of `<IF>` instructions. Each **IF** instruction is the word **IF**, followed by the word **STATE**, by an `<id_state>`, by a `<case_list>`, and by the word **FI**.

`<case_list>` is a list of **CASE** instructions. A **CASE** instruction is the word **CASE**, followed by a `<boolean_expr>`, the word **DO**, a `<list_of_commands>` and the word **DONE**.

If the *current state* is equal to the `<id_state>` specified in the current **IF** instruction, then **CASE** instructions are evaluated. In particular, each time the `<boolean_expr>` associated to a specific **CASE** instruction is true, commands listed in `<list_of_commands>` are executed.

Possible commands listed in `<list_of_commands>` are:

- *Assignment*: A <data>, the symbol "=", a <data> and a ";". A <data> is an <id.name>, a "." (dot) and an <id.attr>. When an assignment command is executed, the value associated to the attribute (<value.attr>) referred to the <data> at the right of the symbol "=", is assigned to the value of the attribute referred to the <data> at the left of the symbol equal. Example: **var1.x = var2.y**; (the value associated to the attribute y of the variable with <var.name> equal to **var2** is assigned to the attribute x of the variable with <var.name> equal to **var1**).
- **PRINT**: the word **PRINT**, followed by a quoted string and a ";". It prints the quoted string.
- **NEW STATE**: the word **NEW STATE** followed by an <id.state> and a ";". This command sets the *current state* to <id.state>. The **NEW STATE** command is the last command of a <list_of_commands>.

<boolean expr> can manage only the following operators: comparison == (equal), != (not equal); logical && (and), || (or), ! (not); and round brackets. Comparison operands can be only a <data> (i.e., <id.name>.<id.attr>) or an unsigned integer number.

Use the parser stack to save the *current state*, and within the **IF** and **CASE** instructions use inherited attributes to decide if execute or not the commands listed in <list_of_commands>.

Goals

The translator must execute the language previously described.

Example

Input:

```
$hello; // <token2>
2017/07/02#2017/09/02-2017/09/18; // <token1>
@%@@%; // <token3>
$1011; // <token2>
%%%%@&@&-13; // <token3>
```

Output:

```
Min: 0 Max: 2
Min: 0 Max: 4
Min: 4 Max: 15
Go to box
Move to box
```

```
*****
mo = monkey { x=1; y=2; z=0; }; // print "Min: 0 Max: 2"
bo = box { x=4; y = 4; z= 0; }; // print "Min: 0 Max: 4"
ba = banana {x=10; y=15; z=4; }; // print "Min: 4 Max: 15"
*****
START STATE S1; // Sets the current state to S1

// The monkey goes to the box
IF STATE S1 CASE mo.x == bo.x && mo.y == bo.y && ( bo.x != ba.x || bo.y != ba.y ) DO // Not executed
    PRINT "Don't move";
    NEW STATE S2;
    DONE
CASE mo.x == bo.x && mo.y == bo.y && bo.x == ba.x && bo.y == ba.y DO // Not executed
    PRINT "Monkey and box are under the banana";
    NEW STATE S3;
    DONE
CASE mo.x != bo.x || mo.y != bo.y DO // Executed
    PRINT "Go to box";
    mo.x = bo.x; mo.y = bo.y;
    NEW STATE S2; // Sets the current state to S2
    DONE
FI

// The monkey moves the box
IF STATE S2 CASE bo.x == ba.x && bo.y == ba.y DO // Not executed
    PRINT "Don't move the box";
    NEW STATE S3;
    DONE
CASE bo.x != ba.x || bo.y != ba.y DO // Executed
    PRINT "Move the box";
    bo.x = ba.x; bo.y = ba.y;
    mo.x = ba.x; mo.y = mo.y;
    NEW STATE S3; // Sets the current state to S3
    DONE
FI
```