Formal Languages and Compilers

21 June 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 three sections: *header*, *states* and *transitions* 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 ";":

- <code> starts with a word composed of at least 6 characters in the set "x" , "y" or "z", disposed in any order and in even number (e.g., xyzxxz, xyzxyzxxzz). The word is followed by the pipe character "|", and optionally by a hexadecimal and odd number between $-3B$ and $aB5$. Remember that odd hexadecimal numbers are those ending with 1, 3, 5, 7, 9, B or b, D or d and F or f.
- <hour>: is an hour with the format "HH:MM:SS" between 10:11:12 and 15:36:47.
- <number> is composed of 4 or 6 binary numbers each of which is composed of 3 or 15 digits. Numbers are separated by means of the characters ".", "-" or "+".

Header section: grammar

Two sequences of tokens are possible in the *header* grammar:

1. **at least 2** <hour> tokens followed by an **odd** number of <code> tokens, which are in turn **optionally** followed by **2 or 4** <number> tokens.
2. only <code> and <number> tokens, which can appear in **any order**. <code> token must appear exactly **one** time, while <number> token exactly **3** times.

States section: grammar and semantic

The *state section* is a list of <states>. The number of <states> is **odd** and **at least 3**. Each element of the list is composed of a <state_name> (an uppercase letter, followed by any number of uppercase letters or numbers or characters "_"), followed by "=", by a "[", a non-empty list of <attributes> in which elements are separated by commas, a "]" and a ";". A <attribute> is composed of a <attribute_name> (one or more lowercase letters), an "=" and a <signed_integer> number.

At the end of this section, the translator must have filled a hash table that contains all the information needed in the next section. **The hash table is the only global variable allowed in all the examination, and it must only contain the information reported in the *state section* (i.e., after this section its content cannot be written, but only read). Solutions using other global variables will not be accepted.**

Transitions section: grammar and semantic

The *transitions section* starts with the command `INIT` followed by a <state_name> and ended with ";" or with the command `DEFAULT;`. The command `INIT` sets the *current state* to <state_name>, while the `DEFAULT;` command sets the *current state* to `S0`.

The first command is followed by a **non-empty** list of `WHEN` commands. A `WHEN` command is the word "WHEN", followed by a <boolean_expr>, the word "DO", a <list_of_case_print_commands>, the word "DONE" and a ";".

<boolean_expr> can manage only the following operators: comparison (`==`), logical (`&&` (and), `||` (or), `!` (not)), and round brackets. The comparison operator can be applied only between a <state_name_and_attribute> and a signed integer number. <state_name_and_attribute> is a <state_name>, followed by a "." and by a <attribute_name>. The value of a <state_name_and_attribute> can be accessed through the global hash table. An example of <boolean_expr> is: `S1.a==1 && !( S1.b==3 || !  S2.a==5)`.

<list_of_case_print_commands> is a **non-empty** list of two kind of commands in a **not predefined order**: `PRINT` and/or `CASE`. Such commands are executed only if <boolean_expr> is TRUE. The `PRINT` command is the word "PRINT" followed by a quoted string and a ";". It prints the quoted string.

The `CASE` command has the following grammar:

`CASE` <state_name>$_1$ `NEXT` <state_name>$_2$ `;`

If <state_name>$_1$ is equal to the *current state*, the command `CASE` sets the *current state* to <state_name>$_2$ and prints <state_name>$_2$ into the screen. Otherwise, if <state_name>$_1$ is not equal to the *current state*, the `CASE` command does nothing.

To manage the value of the *current state* use the parser stack and inherited attributes, in order to check in the `CASE` command if the *current state* is equal to <state_name>$_1$. You can decide in your solution if more than one *current state* change is allowed within a `WHEN` command, or you can make the assumption the only one *current state* change is possible inside each `WHEN` command.

Example

Input:

```
/* Header section (First type of grammar) */
10:11:12;                        /* <hour> */
13:52:58;                        /* <hour> */
xxxxyyyy|-2b ;                   /* <code> */
xyzxxxyyxx|;                     /* <code> */
xxyyzz|223;                      /* <code> */
101.111-101001111100000-101 ;  /* <number> */
111+000+101+010+110-001;         /* <number> */
##
/* States section */
S0 = [ a = 1, b = 2 ];
S1 = [ a = 3, b = 4 ];
S2 = [ a = 5, b = 6, c = 7 ];
##
/* Transitions section */
INIT S2;                       /* current state is set to S2 */
WHEN S0.a==1 || S0.b==2 && S1.a==2 DO   /* TRUE OR TRUE AND FALSE = TRUE OR FALSE = TRUE */
  CASE S1 NEXT S2;             /* Not executed because current state is S2 */
  CASE S2 NEXT S3;             /* Executed, current state is set to S3 */
  PRINT "FIRST WHEN";          /* It prints "FIRST WHEN" in the screen */
DONE;
WHEN ! ( S1.a==5 || S1.b==4) DO        /* NOT ( FALSE OR TRUE) = NOT TRUE = FALSE */
  CASE S1 NEXT S2;             /* Instruction not executed because boolean_expr is FALSE */
DONE;
WHEN ! ! S2.a == 5 DO          /* NOT NOT TRUE = NOT FALSE = TRUE */
  PRINT "THIRD WHEN";          /* It prints "THIRD WHEN" in the screen */
  CASE S3 NEXT S1;             /* Executed, current state is set to S1 */
  CASE S1 NEXT S2;             /* Not executed, because for now current state is S3 */
DONE;
```

Output:

```
S3
FIRST WHEN
THIRD WHEN
S1
```