Formal Languages and Compilers

14 July 2025

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 *instructions* 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 ";":

- `<tok1>`: It is composed of the character "X", a "`>`", and 3 or 7 hexadecimal numbers of 2, 4, or 7 characters. The hexadecimal numbers are separated by the character "#". The first part of the token is optionally followed by a "-" and an IP address (four integers ranging from 0 to 255 and separated by a "."). Example: `X>12#fedc#1234567`.
- `<tok2>`: It is composed of the character "Y", a "`>`", and by 4, 9, or 23 repetitions of odd numbers between -121 and 377. Numbers are separated by a "`*`" or by a "`/`". Example: `Y>-115*3/150*271`.
- `<tok3>`: It is composed of the character "Z", a "`>`", and a hour in the format `HH:MM:SS` between "06:12:38" and "21:31:26". The hour is optionally followed by an even number of repetitions, at least 4, of the words "`xx`", "`yy`", and "`zz`". Example: `Z>11:11:11xxyyxxzz`.

Header section: grammar

In the *header* section the 3 tokens can appear in two ways:

1. at **least 4** (i.e., 4, 5, 6,...) repetitions of `<tok1>`, followed by **3 or 9 or 10** repetitions of `<tok2>`.
2. **from 1 or 3** `<tok2>`, and **any number** of `<tok1>` and `<tok3>` (**even 0**) in any position of the sequence except for the first. This sequence **must start** with a `<tok2>`, the other repetitions of `<tok2>` can appear in **any position** of the sequence.

Instructions section: grammar and semantic

The *instructions* section is composed of a list of **at least 5** `<instr>` in **odd** number (i.e., 5, 7, 9,...).

The four types of instructions are `exec`, `max`, `ass`, and `if`.

Each instruction is terminated by the ";" character, and it has the following syntax:

- `exec <bool_exp>`: executes a boolean expression `<bool_exp>`, prints and returns the result. A `<bool_exp>` can contain the following logical operators: `&` (and), `|` (or), `!` (not), and round brackets to define the scope. Operands are `<var>` (regular expression of a *C identifier*), whose associated value can be accessed by the symbol table (see later), or the constants `T` (true) or `F` (false).
- `max <uint_list>`: it always returns `T` and prints the maximum of the *unsigned integer* values listed in `<uint_list>` (numbers are separated with commas).
- `if <uint1> <uint2>`: based on the comparison of the results returned by the previous two instructions (i.e., r_{-1} and r_{-2}), prints `<uint1>` or `<uint2>` (both are *unsigned integer* numbers) and returns `T` or `F`. In particular, if both $r_{-1} = T$ and $r_{-2} = T$ it prints `<uint1>` and returns `T`, otherwise it prints `<uint2>` and returns `F`. Use inherited attribute to access r_{-1} and r_{-2}.
- `ass <var_list>`: it always returns `T` and assigns to each `<var>` included in `<var_list>` and separated by commas the value returned by the last instruction (i.e., r_{-1}). Use inherited attribute to access r_{-1}. The `<var>` name and the associated boolean value can be stored in a symbol table with `<var>` as key. The content of the symbol table can be used in the `exec` instruction. **This symbol table is the only global data structure allowed in all the examination, and it can be written only in this section.**

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:

```
Y>-3*5*7/11;                    [** tok2 **]
X>2A#12ef#abcd-127.0.0.1;       [** tok1 **]
Z>10:14:12xxyyzzxxyyzz;         [** tok3 **]
##
exec !T & F | F;  [** F & F | F = F | F = F **]
max 4 7 2;  [** print 7 T **]
if 4 5;     [** print 5 F (because r-1=T and r-2=F) **]
ass a b c;  [** a=F b=F c=F T **]
exec a | T; [** F | T = T **]
if 3 7      [** 3 T (because r-1=T and r-2=T) **]
ass d;      [** d=T T **]
```

Output:

```
F
7 T
5 F
T
T
3 T
T
```

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