Compiler Construction (CSCI 355) exercise list 2

Hans de Nivelle

Due: 11.09.2020 (Friday), at 6.00PM

In this exercise, you will build a simple tokenizer using JFlex. Java code for running the tokenizer is provided. JFlex uses the transformation from regular expressions to DFAs, that was taught in CSCI 272.

JFlex and its documentation can be found on `www.jflex.de/` It is normally used in combination with the parser generator tool CUP. We will cover CUP later in the lecture.

Both JFflex and CUP assume that the compiler is written in Java. In the slides, we defined tokens as pairs of form (λ, a), where the type of a depends on the value of λ. In JFlex and CUP, tokens are represented by a class `Symbol`, which has the following fields:

```
int sym;        // Label of the token.
int left;       // Token's line number in input.
int right;      // Token's column position in input.
Object value;   // The attribute.
```

The `Symbol` class is automatically created by CUP based on list of possible tokens. You don't need to worry about this during this exercise. It is already there. CUP checks at runtime that the type of `value` fits to `sym`, while JFlex doesn't.

Download the materials from the wordpress page. It contains a file `stokenizer.jflex`, which already contains a few predefined tokens. Your task is to get JFlex working, to refresh your Java knowledge, and to add a few more tokens.

If you run JFlex, it will create a file **Lexer.java**, which contains an implementation of a DFA-based tokenizer based on the regular expressions that you provided. In addition to the regular expressions, you can provide Java code that computes attributes. This code is copied into **Lexer.java**, so that it will be called when the corresponding regular expression is recognized.

The file **sym.java** (which was constructed by CUP) contains definitions of possible token labels.

If everything goes well, you can compile `Lexer.java`, and have a working tokenizer. File **tokenizer.flex** already contains a few token definitions.

1. Create a regular expression for integers and add it **tokenizer.flex**. You will need to add code that constructs the attribute, here it is:

```
return symbol( sym.INTEGER,
               new list.Integer( new java.math.BigInteger( yytext( ) )));
```

2. Create a regular expression for floating point numbers. In order to create the attribute, you can use

   ```
   return symbol( sym.DOUBLE,
                  new list.Double( new java.lang.Double( yytext( ) )));
   ```

   Examples of floating point numbers that must be accepted are:
   `44, 44.0, -1.45, 4.4E01, 2.99792458e8, 6.67430e-11`

3. We want to generalize the `String` type, because we want to use it for representing operators, like for example `+,-` or `*`. For simplicity, we assume that an operator can be any finite string of symbols built up from the following characters: `- + * / < > = ! | \ / ^ & ~`

   Create a regular expression for such strings, and add it to **tokenizer.jflex**. In order to create the attribute, use

   ```
   return symbol( sym. STRING, new list.String( yytext( )));
   ```

4. In task **nr01**, you defined a regular expression for *C*-style comments that start with `/*` and end with `*/`.

   Add it to **tokenizer.jflex** with action `{ }`. (This means that the tokenizer restarts after having read the comment.)

5. Same for *C*-style comments that start with `//` and last until the end of the line.