Language Processors Lab Week 3

In this lab we will continue practising with JavaCC and lexical specifications. You will learn how to introduce new tokens and how to mix JavaCC with your own Java code.

The LexInt.jj file

Start a Unix shell window and move to your LanguageProcessors directory. Create a new directory `lab3` and move inside it. Download the file `LexInt.jj` from CitySpace (week3) - save it to `lab3`. Make sure you load java and javacc by executing the command:

```
module add java javacc.
```

Start the text editor and load LexTest.jj. Inspect its contents: you will see that the JavaCC specification defines the classes `ParseIntException` and `LexInt`, with an additional static method `parseLiteral`.

Execute JavaCC, compile the generated java files and execute the program by typing the usual commands:

```
javacc LexInt.jj
javac *.java
java LexInt
```

Test the program with a few examples.

Using a text file as input

You can also test your program by re-directing standard input to a text file. Do the following:

- Create a new file `test.txt` using the text editor. Enter a few lines, for example:
  12 a 00306
  b ab 6
- Save the file and type from the command line:

  ```
  java LexInt < test.txt
  ```

Adding new Tokens

- A binary literal consists of the leading characters b or B followed by one or more digits 0 or 1 e.g. `b101, B0001, b111101`. Add the token `BINARY_LITERAL` that implements binary literals.

  Solution: Add a new token to the lexical spefication:

  ```
  TOKEN : /* Integers literals */
  {
    < INTEGER_LITERAL: "0" | (["1"-"9"] (<DIGIT>)*) >
  |
    < BINARY_LITERAL: ("b" | "B") (["0"-"1"])+  >
  }
  ```

  and update the grammar:

```
void TokenList() :
{Token t;}
{
   (
     (t = <INTEGER_LITERAL> |  t = <IDENTIFIER> | t=<REAL> |
      t = <BINARY_LITERAL>)
           { System.out.print("token found: "+ tokenImage[t.kind]);
             System.out.println(" (‘"+t.image+"’)");
   }
   )* <EOF>
}
```

- Introduce a new token that implements single line comments.
  Hint: Use the ~ operator.

Manipulating `Token`

Each token generated by the lexical analyser has a `Token` object associated with it. We can assign this object to a variable e.g. `Token t` and use its contents in the Java code attached to the JavaCC specification. We are currently doing that inside `TokenList`:

- `tokenImage[t.kind])` extracts the token's type.
- `t.image` extracts the token's content - it returns a string.

In particular, the value associated to our `INTEGER_LITERAL` token is a string. In order to get the integer value we need to evaluate the string e.g. the value of string '12' must be converted to 12. The `parseLiteral` method does the job.

Check the implementation of `parseLiteral` and update `TokenList`:

```
void TokenList() :
{Token t; int val;}
{
   ( { val = 0;}
     (t = <INTEGER_LITERAL> { val = LexInt.parseLiteral(t.image,10); }
      |  t = <IDENTIFIER> | t=<REAL> |
      t = <BINARY_LITERAL> | t = <COMMENT>)
           { System.out.print("token found: "+ tokenImage[t.kind]);
             System.out.print(" (‘"+t.image+"’)");
             System.out.println(" value = "+val);
   }
   )* <EOF>
}
```

Inspect the new code. How can we do the same for the binary literals?