Language Processors Lab Week 2 - Solution

The implementation of the solution to lab2 can be found in CitySpace; the file name if `LexTestSolution.jj`.

Regular expressions and JavaCC

New integer definition

Question: Modify the regular expression `INTEGER_LITERAL` to instead match integers that do not start with 0, unless they are zero. For example, it should not accept 012, 00, etc. Instead, for example, the string 0012056 should be split into:
`INTEGER_LITERAL('0')`, `INTEGER_LITERAL('0')`, `INTEGER_LITERAL('12056')`.

- The regular expression: 0 | ([1-9] [0-9]*)

- The implementation:

```
TOKEN : /* Literal integers. */
{
  < INTEGER_LITERAL: "0" | (["1"-"9"] (<DIGIT>)*) >
}
```

where `<DIGIT>` is defined by:

```
TOKEN : /* Definitions for use in other lexical definitions
           Token names must start with '#'. */
{
  < #DIGIT: [ "0"-"9"] >
}
```

Note that in JavaCC all characters or strings need to be put between double quotes. Regular expression operators, such as `?`, `*` and `+`, need to be applied to expressions sorrounded by parenthesis e.g. `<DIGIT>*` won't work, `(<DIGIT>)*` should be used instead.

Adding a new token: `<KEYENDWHILE>`

Question: Add a new keyword to match `'endwhile'`. Note that you will also have to modify the grammar i.e. TokenList.

- The regular expression: `endwhile` or `"endwhile"`
- The implementation:

```
TOKEN : { < KEYENDWHILE: "endwhile" > }   /* lab2 - new */
```

However, just adding the new token to the JavaCC definition won't be enough. The JavaCC file contains both a lexical and syntactic specification (grammar). In our example, the grammar is specified by the non-terminal `TokenList` at the end of the file. The grammar is basically a Kleene-closure (*) that encloses a list of token names and alternation operators |. It basically says: accept a sequence of the tokens specified here.

In order to instruct the parser to accpt out new token we must re-write the definition of `TokenList` to:

```
void TokenList() :
{Token t;}
{
   (
     (t = <KEYWHILE> | t = <INTEGER_LITERAL> | t = <KEYTRUE> |
      t = <KEYFALSE> |  t = <IDENTIFIER> |
      /* lab2: New tokens added here */
      t = <KEYENDWHILE> | t=<REAL> )
           { System.out.println ("token found: "+ tokenImage[t.kind]+
                                    " (‘"+t.image+"’)"); }
   )* <EOF>
}
```

Note that I have already included `<REAL>`.

The `<REAL>` token

Question: Add the regular expression definition of the `REAL` token to instead match signed real numbers as written in Pascal. Such numbers must contain a decimal point, and at least one digit before and after the decimal point. They may optionally be prefixed by '-' and may optionally be followed by a signed exponent that begins with the letter 'E' and is followed by a (possibly signed) integer. In this notation, 39.37, -6.336E4, 0.894E-4 and 0.0 are legal, while .36, 4. and +.7E6 are illegal.

- Regular expression:
  -? (0 | [1-9][0-9]*) . [0-9]+ ((e|E) -? (0|[1-9][0-9]*))?

  As expressions get more complicated, is becomes useful to introduce intermediate names, as the ones used by JavaCC (`#` prefix). The expression above can be re-writen as:

  `<integer>` -? (0 | [1-9][0-9]*) . [0-9]+ `<exponent>?`
  `<exponent>` (e|E) -? (0 | [1-9][0-9]*)

- Implementation:

  ```
  TOKEN : /* Literal integers. */
  {
    < REAL: ("-")? ("0" | (["1"-"9"] (<DIGIT>)*)) "." (<DIGIT>)+
            <EXPONENT> >
  }
  ```

  where:

  ```
  TOKEN :
  {
    < #EXPONENT: (("e" | "E") ("-")? ("0" | (["1"-"9"] (<DIGIT>)*)))? >
  }
  ```

Testing from a file

In Unix, standard input (default is the keybord) can be re-directed to a file by running the program with `< filename`. For example, in order for our program to accept input form a file we can write:

```
java LexTest < input.txt
```

where `input.txt` is a text file.