Language Processors Lab 3 - 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 Moodle (Lab3) to your `lab3` directory. Make sure you load java and javacc by executing the command:

```
module add java javacc.
```

Start the text editor and load `LexInt.jj`. Inspect its contents: you will see that the JavaCC specification defines the `LexInt` class followed by three tokens: `SKIP`, `INTEGER_LITERAL` and `IDENTIFIER`.

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

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

Test the program with a few examples. You will see that the lexical analyser acepts unsigned integers and identifiers that start with a letter. The program prints the type and contents of each accepted token, followed by its numeric value (cuurently undefined).

Modifying Token Specifications

- `IDENTIFIER`: Most programming languages accept identifiers that contain special characters such as underscore(`'_'`). `LexInt` does not accept underscores as part of indentifiers; verifiy this by typing a few examples. Modify the `IDENTIFIER` token specification so that it includes underscores (after the first letter).

  Solution: Replace the current specification with:

  ```
  TOKEN : /* Identifiers. */
  {
    < IDENTIFIER: <LETTER> (<LETTER> | <DIGIT> | "_")* >
  }
  ```

- `INTEGER_LITERAL`: The current specification for integers in `LexInt` accepts strings with leading zeroes e.g. `0034`. Furthermore, it only deals with unsigned integers i.e. it does not accept a sign symbol (plus or minus) e.g. enter `-45` and see what happens. Modify the current `INTEGER_LITERAL` specification so it accepts integers with (optional) signs and no leading zeroes.

  Solution: The new `INTEGER_LITERAL` specification should look like this:

  ```
  < INTEGER_LITERAL: ("+"|"-")? ("0" | ["1"-"9"] (<DIGIT>)*)  >
  ```

  Explanation: The regular expression above indicates that an integer literal may start with a plus or minus (the alternation `|` symbol means OR). The sign symbol is optional, as indicated by the question mark. We want to make sure that integers do not contain leading zeroes. Therefore, if an integer starts with zero then it must be `"0"`. Otherwise, it must start with any digit between 1 and 9 (i.e. non-zero), followed by a possibly empty sequence of digits.

Adding new Tokens

- Keywords and reserved words. A keyword is a word or identifier that has a particular meaning to a programming language. Example of keywords are words used in control flow such as `if`, `for` and `while`. Usually keywords are reserved words and cannot be used for variable or function names. Define a new token `IF` that specifies the keyword `if`

  Solution: Add the following token specification AFTER (below) the `IDENTIFIER>` token specification:

  ```
  TOKEN : /* Reserved words and Keywords */
  {
    < IF: "if" >
  }
  ```

  and update the grammar specification at the bottom of the document to:

  ```
  (t = <INTEGER_LITERAL> | t = <IDENTIFIER> | t=<IF> )
  ```

  Run javacc. The following warning should appear:

  ```
  Warning:  "if" cannot be matched as a string literal token at line 45,
  column 3. It will be matched as  <IDENTIFIER>.
  ```

  Ignore the warning message, compile the program and execute it. Enter the following input: `"if25 if 25"`. What happens? The first word is correctly identified as an identifier. Unfortunalety, the second word, `if`, should have been identified as an `IF` token instead of an identifier. This is related to the warning displayed above. JavaCC, when presented with a string that matches two tokens, reports the token that has been defined first (in this case, IDENTIFIER).

  Solution: Move the `IF` specification BEFORE (above) the IDENTIFIER specification. Run javacc, re-compile and execute the program with the same input. What happens? `if` is now recognised as an `IF` token. Note that `if25` could have been split into an IF token and an integer. However, Javacc will always try to match the longest possible string against the available 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 accordingly.

- Comments and the complement (`~`) operator: The complement `~(e)` operator, as with set theory, is used to express the characters not belonging to the set specified by `e` i.e. its negation. For example, `~["a","b"]` expresses all characters except `a` and `b`.
  Use the complement operator to define the token `<TOKEN>` that implements single line comments that start with `//`. Single line comment must end with the new line character.

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 `Integer.parseInt` method does the job. We need to do something similar for binary numbers e.g. string b110 has value 6. How do we do this? Fortunately, `Integer.parseInt` can take the base or radix of a number as an argument. In our case, we need to pass 2 as argument: `Integer.parseInt("110",2)`. Note that we have to remove the prefix!!

Update `TokenList` with the following code (this assumes you have defined COMMENT):

```
(
  (t = <INTEGER_LITERAL> | t = <IDENTIFIER> | t=<IF> |
   t = <BINARY_LITERAL> | t = <COMMENT> )
      { System.out.print("token found: "+ tokenImage[t.kind]);
        System.out.print(" (‘"+t.image+"’) value: ");
        switch (t.kind) {
        case INTEGER_LITERAL:
          System.out.println(Integer.parseInt(t.image)); break;
        case BINARY_LITERAL:
          s = t.image.substring(1);
          System.out.println(Integer.parseInt(s,2)); break;
        default: System.out.println("UNDEFINED");
      }
    }
)* <EOF>
```

Note that we have removed the first character of a binary string. We have used the `substring` method from the `String` class.

Exercise: Define a new token `RATIONAL` that implements rational numbers of the form `1/2 3/5 23/100 0/5` etc. How do we compute their numeric value?