Language Processors Lab Week 3 - Solution comments

The solution of Lab3 has been uploaded to CitySpace. This documentsclarifies some of the trickiest points.

1 Parsing Integer Literals

Lab 3 introduces a method that translates integer literals (decimal, binary, octal and hexadecimal) from a string representation into their corresponding decimal value. In other words, it translates a string that represents numbers written in a particular base (10,2,8 and 16, respectively) into their integer value.

For example, binary number b101 (base 2) has value 5 = 1+0*2+1*4 and decimal literal 375 (base 10) has value 375=5+7*10+3*100. More examples are shown in the table below:

String	Base	Value
1100	2	$10 = 0 + 0 * 2 + 1 * (2^2) + 1 * (2^3)$
1100	8	$576 = 0 + 0 * 8 + 1 * (8^2) + 1 * (8^3)$
101	8	$65 = 1 + 1 * (8^2)$
101	16	$257 = 1 + 1 * (16^2)$
50	16	$80 = 0 + 5 * (16)$
F0A	16	$3850 = 10 + 15 * (16^2)$

Recall that hexadecinal numbers use characters '0'-'9', 'a'-'f' and 'A'-'F' where 'A'/'a' denote 10, and 'f'/'F' denote 15. Thus, given base b and string

$$x = \bar{x}_{(n-1)} \ldots \bar{x}_i \ldots \bar{x}_1 \bar{x}_0$$

where $\bar{x}_i$ denotes a single character, the value of x is defined as follows:

$$val(x) = x_0 + x_1 * b + \ldots + x_i * b^i + \ldots X_{(n-1)} * b^{(n-1)}$$

assuming that x is the numerical value of character $\bar{x}$ e.g. if $\bar{x}$ is '5' then x is 5. This can be easily implemented in Java. For example, the method `parseLiteral2` below transforms a integer literal `lit` written in base `base` into an integer:

```
public static int parseLiteral2(String lit, int base)
                                                throws Exception {
  int exp, valchar=0, val=0, l;
  char ch;
  StringBuffer buf = new StringBuffer(lit);
  l = buf.length();
  exp = 1;
  for(int i=l-1; i >= 0; i--) {
    ch=buf.charAt(i);
    if (ch >= '0' && ch <= '9')
      valchar = ch-'0';
    else if (ch >= 'a' && ch <= 'f')
```

```
      valchar = 10 + (ch-'a');
    else if (ch >= 'A' && ch <= 'F')
      valchar = 10 + (ch-'A');
    else
      throw new ParseIntException("Bad Char "+ch);
  val = val + valchar*exp;
  exp = exp*base;
  }
  return val;
}
```

Note that we can obtain the value of each character by subtracting '0' from it. Hexadecimals greater than 9 need a special case e.g. 10 + (ch-'a') or 10 + (ch - 'A'). Another implementation follows:

```
public static int parseLiteral(String lit, int base)  {
  int valchar=0, val=0, l;
  char ch;
  StringBuffer buf = new StringBuffer(lit);
  l = buf.length();
  for(int i=0; i < l; i++) {
    ch=buf.charAt(i);
    if (ch >= '0' && ch <= '9')
      valchar = ch-'0';
    else if (ch >= 'a' && ch <= 'f')
      valchar = 10 + (ch-'a');
    else if (ch >= 'A' && ch <= 'F')
      valchar = 10 + (ch-'A');
    else {
      System.out.println("Bad Char "+ch); // better trow an exception
      return -1;
    }
    if (valchar >= base) {
       System.out.println("Bad Char "+ch+" greater or equal than base "+base);
       return -1;
    }
    val = val*base + valchar;
    }
  return val;
}
```

which exploits the fact that

$$val(x) = x_0 + b * (x_1 + b * (x_2 + \ldots + b * (x_{n-2} + b * x_{n-1}) \ldots))$$

The above implementation is the one included in Lab3's code.

Note: The current implementation of parseLiteral is limited by the size of the Java type `int`. This means that we will get wrong results with big numbers. The code can be improved by using `long` instead.

2 INTEGER_LITERAL: Manipulating `Token`

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`. We introduce binary literals to our lexical analyser with the addition of new token `BINARY_LITERAL` to the JavaCC specification. This is carried out in two steps. First, we need to add a new token specification for `BINARY_LITERAL_`. We can do this by adding a couple of lines to the token declaration used to define `INTEGER_LITERAL`:

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

The scanner now recognises the new token `BINARY_LITERAL`. However, the parser does not know that it has to match the input against this new token. This is achieved by adding the new token name to the grammar specified by `TokenList`:

```
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>
}
```

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` object.

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 will do the job. We need to insert Java code that must be executed right after the parser matches `INTEGER_LITERAL` which calls `parseLiteral` with the contents of the token:

```
void TokenList() :
{Token t; int val;}
```

```
{
   ( { val = 0;}
            // NEW CODE after the token match
     (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>
}
```

We also want to compute the numeric value of binary literals. Fortunately `parseLiteral` works for binary numbers as well. However, it will refuse to take as input the string stored by `t.image`. Why? Because our binary strings start with b or B, and `parseLiteral`, when working with base 2 numbers, will only accept 0s and 1s. Thus, we need to strip off that first character from `t.image`. That's easy - we just need to call the `String` class `substring` method and call `parseLiteral` with the appopiate base (i.e. 2):

```
val = LexInt.parseLiteral(t.image.substring(1),2);
```

The new code should like like this:

```
void TokenList() :
{Token t; int val;}
{
   ( { val = 0;}
     (t = <INTEGER_LITERAL> { val = LexInt.parseLiteral(t.image,10); }
      |  t = <IDENTIFIER> | t=<REAL> |
      t = <BINARY_LITERAL>  { val = LexInt.parseLiteral(t.image.substring(1),2); }
      | t = <COMMENT>)
           { System.out.print("token found: "+ tokenImage[t.kind]);
             System.out.print(" (`"+t.image+"')");
             System.out.println(" value = "+val);
   }
   )* <EOF>
}
```

If we want to print values only for Tokens that denote integers, we can change the body of `TokenList` to the following:

```
void TokenList() :
{Token t; int val;}
{
   ( { val = 0;}
     (t = <INTEGER_LITERAL> |  t = <IDENTIFIER> | t=<REAL> |
```

```
      t = <BINARY_LITERAL> | t = <COMMENT>)
      { System.out.print(tokenImage[t.kind]+" (`"+t.image+"') ");
        switch(t.kind)  {
        case INTEGER_LITERAL:
           val = LexInt.parseLiteral(t.image,10);
           System.out.println(" value = "+val);
           break;
        case BINARY_LITERAL:
           val = LexInt.parseLiteral(t.image.substring(1),2);
           System.out.println(" value = "+val);
           break;
        default:
           System.out.println(" "); // new line
        }
      }
   )* <EOF>
}
```

IMPORTANT: Check the `String` Java class specification.

3 Single Line Comments

Comments are used widely by programming languages. They allow the programmer to introduce text at any point of the program that will ultimately be ignored by the compiler. Single line comments start with a special reserved keyword .e.g `"//"` followed by a sequence of characters (usually of any type) ended by the newline character. Thus, they can only run for a single line.

An example of a single line comment specification in JavaCC:

```
TOKEN :
{
 < COMMENT: "//" (~["\n","\r"])* ["\n","\r"] >
}
```