Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Lab 2: Lexer Implementation

Learning Objectives Upon successful completion, students will be able to:

- understand the issues in handling different types of tokens;
- implement lexers for typical tokens of a programming language, both manually and with JavaCC.

Preparation

You can either download the file `lab2.zip` from D2L to your laptop, or if you are on the CS linuxlab system, copy the file `/u/li/321/lab2.zip` or `/u/li/321/lab2.tar` to your directory. After unzipping (or untarring) the file, enter the directory `lab2`. You'll see a sequence of sub-directories, 01-06. Each directory contains two lexers for the same set of token definitions. `Lexer1.java` is a manual version, while `Lexer2.jj` is a JavaCC version, which can be compiled to a Java program, `Lexer2.java`, with the `javacc` compiler. There are two driver programs for testing these lexers, `RunLexer1.java` and `RunLexer2.java`. A `Makefile` is provided for compiling these programs.

The exercises that are marked with an asterisk ($\star$) are a little more challenging. If you get stuck on one, you should move on to the next unit, and revisit it later when you have more time (*e.g.* after the lab).

01. The RE Language's Tokens

The lexers in this directory are for the RE Language from Lab 1. `Lexer1.java` is a slightly modified version of the `Lexer.java` program from Lab 1: The individually defined integer token codes are replaced with a Java enumeration. The enumeration representation has some advantages. (1) Token code can now be associated with a user-chosen name for printing out. (*e.g.* Instead of showing an integer code `1`, we can now show `ID`.) (2) All the token codes are now in a single collection, and can be processed together using an enumerator.

`Lexer2.jj` contains a JavaCC specification for the RE tokens:

"from 01/Lexer2.jj"

```
PARSER_BEGIN(Lexer2)
public class Lexer2 {}
PARSER_END(Lexer2)

// Token Definitions
//
SKIP:   { " " | "\t" | "\r" }               // whitespace chars

TOKEN:  { <LETTER: ["a"-"z"]> }
TOKEN:  { <ALTER:  "|"> }
TOKEN:  { <REPEAT: "*"> }
TOKEN:  { <LPAREN: "("> }
TOKEN:  { <RPAREN: ")"> }
TOKEN:  { <END:    "\n"> }
```

It is a straightforward task to write such a specification if you have the tokens' regular expression definitions.

Side Note: In the above JavaCC program, the tokens are defined individually, each with a `TOKEN` keyword. An alternative form is to group them under a single `TOKEN` keyword:

```
TOKEN:
{ <LETTER: ["a"-"z"]> | <ALTER: "|"> | <REPEAT: "*"> | <LPAREN:"("> | <RPAREN: ")"> | <END: "\n"> }
```

Exercises

1. Compile and run both lexers:

   ```
   linux> make lexer1 lexer2        (or just 'make')
   linux> java RunLexer1 test1.in
   linux> java RunLexer2 test1.in
   ```

2. Look inside the program `Lexer1.java`. In the main lexer routine, `nextToken()`, there are three statements: a `while`, an `if`, and a `switch`, in that order. Can these three statements be re-arranged in a different order? If you are not sure, you may want to try it and see the effects.

3. Add three new tokens to the RE language:

   ```
   <DIGIT :  ['0'-'1']>       // single digits
   <ONEMORE: '+'>             // one or more repetition
   <ZEROONE: '?'>             // zero or one occurrence
   ```

   Modify `Lexer1.java` and `Lexer2.jj` so that they can handle the new tokens. Write a new test input to test your new lexers.

   Side Note: If you want to keep the original version of the programs, you could create a new subdirectory, and copy and modify programs in there; or you could give your programs new names, say `Lexer3.java` and `Lexer4.jj`.

02. Operators

The lexers in this directory are for recognizing operators. Unlike the RE tokens, some operators are defined with multiple characters, *e.g.* ==, <=, and >=. So the first thing is to modify the token representation to use a `String` to hold the lexeme:

"from 02/Lexer.java"

```
public class Token {
  TokenCode code;    // code
  String lex;        // lexeme
  int pos;           // position
  ...
}
```

For the manual lexer, the new issue is to handle the common prefix among different token patterns, *e.g.* `"<"` and `"<="`. In class, we learned that we could use lookahead to help distinguish the prefix cases: After seeing a `"<"`, the lexer could peek at the next character, if it is a `"="`, then the token is `LE`, otherwise, it is `LT`.

In `Lexer1.java`, a single-character buffer, `nextC`, is used to hold the next lookahead character. During the initialization of the lexer program, the first character from input is read into `nextC`. After that, each time `nextChar()` is called, the character in `nextC` is returned; and a new character is brought into it:

"from 02/Lexer1.java"

```
//
private static int nextChar() throws Exception {
  int c = nextC;
  nextC = input.read();
  if (c != -1)
    pos++;
  return c;
```

To peek at the next lookahead character, the program simply reads the variable `nextC` directly:

```
"from 02/Lexer1.java"
        return new Token(TokenCode.ASSGN, "=", pos);
      case '<':
        if (nextC == '=') {
          c = nextChar();
          return new Token(TokenCode.LE, "<=", pos);
        }
```

Exercises

1. Compile and run `Lexer1`.
2. Complete the JavaCC specification `Lexer2.jj`. Compile and run it.
3. Add four new operators, <<, <<=, >>, and >>=, to the collection. Modify both lexers to accommodate them. Run and test the new programs (with a new test input).

03. Integer Literals

The lexers here are for recognizing integer literals. An integer literal is typically defined as a non-empty sequence of digits, regardless length. However, to be a *valid* integer token, the value represented by the integer literal must be within the range of the integer type of the language. For instance, for Java's `int` type, the literal's value must be less than 2,147,483,648 (2^{31}).

This brings two issues to the lexer implementation. (1) An integer literal's lexeme can be of arbitrary length. (2) The lexer needs to validate each integer literal after recognizing it.

To handle the first issue, we need to use an *extensible* buffer to hold the lexeme. In `Lexer1.java`, a variable of type `StringBuilder` is introduced for this purpose. A `StringBuilder` string can be appended with new characters and be expanded to arbitrary size. (The class `StringBuffer` has the same properties, but is slightly more expensive to use.)

Note that a `StringBuilder` object is not a `String`. You need to use the `toString()` method to convert it to a `String` object, if needed:

```
"from 03/Lexer1.java"
      if (isDigit(c)) {
        StringBuilder buffer = new StringBuilder();
        buffer.append((char) c);
        while (isDigit(nextC)) {
          c = nextChar();
          buffer.append((char) c);
        }
        String lex = buffer.toString();
```

To handle the second issue, we need to compute an integer literal's value and check it against the integer type's range. This could be done with a brute force binary-to-decimal conversion, but in `Lexer1.java`, we assume the integer literal's limit is the same as Java's, and use Java's library routine `Integer.parseInt()` to do the job:

```
"from 03/Lexer1.java"
        // catch ill-formed literals
        try {
          Integer.parseInt(lex);
          return new Token(TokenCode.INTLIT, lex, pos);
        }
        catch (Exception e) {
          throw new Exception("Lexical Error at column " + pos +
                          ": Invalid integer literal " + lex);
      }
```

For the JavaCC specification, the first issue is handled automatically. The second issue is handled in the same way as in `Lexer1.java`. In the action code for the `INTLIT` token, `Integer.parseInt()` is called:

"from 03/Lexer2.jj"

```
TOKEN:
{ <INTLIT: (["0"-"9"])+> { try { Integer.parseInt(matchedToken.image); }
                           catch (Exception e) {
                             throw new TokenMgrError("Lexical Error at column " +
                                                     matchedToken.beginColumn +
                                                     ": Invalid integer literal " +
                                                     matchedToken.image, 0);
                           }
                         }
```

Side Note: In both of the above code, the Java integer-conversion exception is caught locally, and re-thrown as a lexical exception, to provide a better error description.

Exercises

1. Compile and run both lexers.
2. ($\star$) Add the octal integer form into both lexers. With this change, any sequence of digits with a leading 0 is considered an octal literal.[1] Such a sequence's value needs to be computed using a base of 8. Java's `Integer.parseInt(String literal)` routine is default to the base of 10. However, the routine has a more general version with two parameters, `Integer.parseInt(String literal, int base)`. You need to select the proper version to use depending on the literal's form.

   The change to the JavaCC version is more challenging. This is because in JavaCC, semantic actions can only be appended at the end of a token definition. Since both the decimal and octal literal forms correspond to the same `INTLIT` token, the semantic action for checking literal's value can only be defined for the joint case:

   ```
   TOKEN:
   <INTLIT:  ..Decimal Form.. | ..Octal Form..>
      { ... Java code for checking literal's value ... }
   ```

   This means that information about the literal's form is not explicitly available to the Java code. You need to inspect the lexeme to decide whether it's a decimal or an octal. The lexeme is accessible through the field `matchedToken.image`.

   Compile and test your new lexers.

04. IDs and Keywords

Now we are on to lexers for recognizing IDs and keywords. As discussed in class, the approach for dealing with IDs and keywords in a manual lexer implementation is to treat keywords as IDs first, and then single them out by comparing the buffered lexeme with the list of keywords:

"from 04/Lexer1.java"

```
    // recognize ID and keywords
    if (isLetter(c)) {
      StringBuilder buffer = new StringBuilder();
```

[1] Strictly speaking, a singleton digit `0` is classified as a decimal literal, so that the decimal literals can have a complete value range. To represent the value 0 in octal form, you can write `00`. This does not have any impact on lexer implementation.

```
      buffer.append((char) c);
      while (isLetter(nextC)) {
        c = nextChar();
        buffer.append((char) c);
      }
      String lex = buffer.toString();
      if (lex.equals("if"))
        return new Token(TokenCode.IF, lex, pos);
      if (lex.equals("else"))
        return new Token(TokenCode.ELSE, lex, pos);
      return new Token(TokenCode.ID, lex, pos);
    }
```

Exercises

1. Complete the JavaCC specification `Lexer2.jj`. Compile and run both lexers.
2. ($\star$) In `Lexer1.java`, the searching for a keyword match is done sequentially through a sequence of `if` statements. This is not very efficient when the list of keywords is long. An alternative approach is to store the keywords in a hash table and do a hash lookup when a match needs to searched. Look up online for information on Java's `HashMap` library, and see if you can replace the `if` statements with a `HashMap` lookup.

05. Dealing with Comments

In this last unit, we'd like to address the issues of handling comments. As discussed in class, Most languages support two forms of comments. The following are RE definitions for Java's *single-line* comment and *block* comment:

```
<SL_COMMENTT:  "//" (~['\n',<EOF>])*>
<BLK_COMMENT:  "/*" (~("*/"|<EOF>))* "*/">
```

In other words, a single-line comment starts with `"//"` and ends with the first occurrence of either an `EOL` or an `EOF`. A block comment starts with `"/*"` and ends with the first occurrence of `"*/"`. In both cases, any characters can be included in the comment. In the block comment case, if an `EOF` appears before the end-marker, `"*/"`, it is an "unterminated comment" error.

Side Note: To help to see the lexing process, the two forms of comments are defined as actual tokens in both lexers in this directory.

Exercises

1. Complete the program `Lexer1.java` by filling in the code for recognizing block comments. There shouldn't be a need to change other parts of the program.
2. ($\star$) Try to write a JavaCC specification for Java's block comments without using lexer states. This means that you need to come up with a proper RE definition for the block comments, *i.e.* one without needing to negate strings. (JavaCC's negation operator `"~"` can only be used with single characters.)

06. Error Handling in JavaCC

We are back with the lexers for the RE language. The file `test1.in` contains illegal characters to the RE language. When running the two lexers with `test1.in` as input, you get quite different error messages back:

```
linux> java RunLexer1 test3.in
java.lang.Exception: Lexical Error at column 1: Illegal character #
linux> java RunLexer2 test3.in
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 1.  Encountered: "#" (35),
at Lexer2TokenManager.getNextToken(Lexer2TokenManager.java:267)
at Lexer2.getNextToken(Lexer2.java:128)
at RunLexer2.main(RunLexer2.java:29)
```

The message from `Lexer2` is much cluttered, and there is no direct indication of "illegal character". This can be easily fixed.

Exercises

1. Uncomment the following two lines in `RunLexer2.java`:

   "from 06/RunLexer2.java"
   ```
   //    } catch (TokenMgrError e) {
   //      System.err.println(e);
   ```

   Compile and run again. You'll see a cleaner error message without the stack trace.

2. Uncomment the following lines in `Lexer2.jj`:

   "from 06/Lexer2.jj"
   ```
   //SPECIAL_TOKEN: /* Catch illegal character */
   //{
   //  <~[]> { if (true) throw new TokenMgrError("Illegal character at column " +
   //                                            matchedToken.beginColumn + ": " +
   //                                            matchedToken.image, 0); }
   //}
   ```

   This wildchar pattern, `<~[]>`, will match any character that is not matched by any of the legal token patterns defined above, and give the user the control as to what message to report. Try compile and run to see the effect.

Side Note: If you need further customization, you can define a subclass to the class `TokenMgrError()`, and place it in between of the `PARSER_BEGIN` and `PARSER_END` brackets.