Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @SRTC 155, Labs: Tue 14:00-15:50 & Wed 12:00-13:50 @FAB 88-10

Lab 2: Lexer Implementation

Learning Objectives

Upon successful completion, students will be able to:

- use JavaCC to implement a lexer, and
- manually implement a lexer for simple languages.

Preparation

1. Log in to a Linux Lab machine and check to see whether you have Java 7 or 8 in your environment — type `"which java"`, and see if it shows `jdk1.7` or above. If not, run `addpkg`; select `java8` and save; then log out and log in again to make it functional.
2. We will use JavaCC (Java Compiler Compiler) as the main programming tool for our compiler projects this term. It is already available on the Linux Lab system. However, if you want a copy for your own computer, you may download `javacc-5.0.zip` from D2L.
3. Download from D2L the file `lab2.zip` and unzip it. Enter the directory `lab2`.

Part 1. JavaCC as a Lexer Generator

JavaCC is a combined lexer and parser generator for use with Java applications. It takes both token and grammar specifications and generates a Java program that implements a top-down parser (with an internal lexer). In this lab, we'll practice the use of JavaCC as a lexer generator only. We'll use its parser-generator features later.

A JavaCC program starts with a header section, which declares the parser class:

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

Even though we are using JavaCC as a lexer generator, we still need to include this header section (and use the `PARSER_BEGIN` and `PARSER_END` keywords). Note that the three occurrences of the user-provided `Name` must match. (It's a convention to have the `.jj` file also match the `Name`.)

A `main` method can be defined in the class in the header section:

```
PARSER_BEGIN(Lexer)
public class Lexer {
  public static void main(String [] args) {
    ...
  }
}
PARSER_END(Lexer)
```

This would make the JavaCC-generated lexer a *standalone* program, capable of direct execution. (Without the `main` method, the generated program would be in the form of a library and would need another program to drive it.)

Token definitions follow the header section, each definition takes the form:

```
TOKEN: { <name: pattern> }
```

The `name` is a user-chosen name for the token, which can be omitted if an explicit name for the token is not needed (*e.g.* tokens with unique lexeme strings). The `pattern` is a regular expression in JavaCC notation for defining the token.

JavaCC uses the following notation for regular expressions:

- `""` for quoting literal strings and single characters, *e.g.* `"a"`, `"begin"`.
- `[]` for character selections, *e.g.* `["a"-"z"]`.
- `()+` and `()*` for repetitions, *e.g.* `("a")+`.
- <> for quoting previously-defined patterns.

When there are multiple token definitions, normal regular expression rules apply, *e.g.* "maximal munch" and "first match".

00 (The "echo" program)

This program reads from terminal input and prints every character it reads; effectively implementing the "echo" function. The input source, in this case `System.in`, is passed in as a parameter to the constructor of the main lexer class. The main interface function to the lexer is `getNextToken()`. Each call to it returns a new token:

from 00/Lexer.jj

```
PARSER_BEGIN(Lexer)
public class Lexer {
  public static void main(String args[]) {
    Lexer lexer = new Lexer(System.in);
    Token tkn = lexer.getNextToken();
    while (tkn.kind != 0) {
      System.out.print(tkn.image);
      tkn = lexer.getNextToken();
    }
  }
}
PARSER_END(Lexer)
```

A token is represented by a `Token` object, which has a `kind` field representing the token's internal code and an `image` field holding the token's lexeme. A token's internal code is assigned by JavaCC automatically, based on its position in the token definition section in the `.jj` program. The JavaCC-generated file `Token.java` contains the definition of the `Token` class. Some of its key fields are shown here:

from 00/Token.java

```
public class Token implements java.io.Serializable {
  public int kind;
  public int beginLine;
  public int beginColumn;
  public int endLine;
  public int endColumn;
  public String image;
```

```
  public Token next;
  ...
}
```

In JavaCC, a wildchar is represented by `"~[]"` (literally, the negation of an empty character). It can match *any* character, except the end of file (`EOF`) character. So the token definition

```
TOKEN: { <ANYCHAR: ~[]> }
```

will match any character in the input stream, including the invisible ones such as `"\n"`.

01 (Recognizing IDs)

This program shows the use of another pattern category, `SKIP`. Matched strings for the `SKIP` patterns are simply thrown away. `SKIP` patterns and `TOKEN` patterns can interleave in any way they need to be:

from 01/Lexer.jj

```
SKIP:   { " " | "\t" | "\n" | "\r" }        // white space chars
TOKEN:  { <ID: (["A"-"Z"]|["a"-"z"])+> }    // identifiers
SKIP:   { <OTHER: (~[])> }                  // any other chars
```

But note that the ordering of the three pattern lines in this program are important.

Exercise: Switching the `ID` and `OTHER` pattern lines and see the effect. (If you want to keep the original `Lexer.jj` intact, copy it to `Lexer0.jj` first.)

02 (Token definitions)

This program shows a larger set of token definitions. Note that multiple tokens can be defined in one group. Tokens that start with a # are private tokens; their purpose is to help simplify other token definitions:

from 02/Lexer.jj

```
TOKEN:  // identifiers and integer literals
{
  <#DIGIT:  ["0"-"9"]>            // internal tokens
| <#LETTER: ["A"-"Z"]|["a"-"z"]>
| <ID:      (<LETTER>)+>
| <INTLIT:  (<DIGIT>)+>
}
```

Exercise: Add token definitions for string literals and floating-point literals to `Lexer.jj` and test your new program.

03 (Lexer states)

Finding a regular expression for Java-style multi-line comments is not very easy. JavaCC's lexer states provide a simpler solution to the problem of skipping multi-line comments. With lexer states, we can direct the lexer to enter a new state upon seeing a specific pattern (*e.g.* `"/*"`); have it match tokens with an independent set of matching rules for that state; and have the lexer return to the default state upon seeing another specific pattern (*e.g.* `"*/"`). The key is that the matching rules of different states do not interfere with each other.

from 03/Lexer.jj

```
SKIP :  // comments
{
  <"//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>  // single-line comment
| "/*" : ML_COMMENT            // enter multi-line-comment state
}

<ML_COMMENT> SKIP :            // inside multi-line-comment state
{
  "*/" : DEFAULT               // return to default state
| <~[]>
}
```

Exercise: Try to define a direct `SKIP` pattern for multi-line comments, without using lexer states.

04 (Token attributes)

This program shows how to display useful token attributes and how to augment with new attributes. It also shows how to insert semantic actions into patterns.

As an example, suppose we want to display symbolic names for some tokens, *e.g.* `ID` and `INTLIT`, instead of just their internal code. The problem is that we can not modify the `Token` class definition. An *ad hoc* solution is to define a static class variable in the main lexer class, and have each involved token pattern assign its token name to the variable before returning from match:

from 04/Lexer.jj

```
public class Lexer {
  static String tknName = null;   // an ad hoc token name solution
  ...
}
PARSER_END(Lexer)

TOKEN:  // identifiers and integer literals
{
  <#DIGIT:  ["0"-"9"]>
| <#LETTER: ["A"-"Z"]|["a"-"z"]>
| <ID:      (<LETTER>)+> { Lexer.tknName = "ID"; }
| <INTLIT:  (<DIGIT>)+>  { Lexer.tknName = "INTLIT"; }
}
```

05 (Token values)

For tokens with value attributes, such as integer literals, floating-point literals, and Boolean literals, the lexer has a need to convert their lexeme sequences into proper value forms. This program shows an example of converting a sequence of digits to an integer value. While it is possible to do it manually one digit at a time, it is more convenient to a Java utility function `Integer.parseInt()` to do it:

from 05/Lexer.jj

```
    if (tkn.kind == INTLIT)
      System.out.println(Integer.parseInt(tkn.image));
```

Exercise: Define a token pattern for Java Unicode escape sequences, *e.g.* `"\u...uXXXX"` (multiple `u`s allowed; each `X` is a hexadecimal character), and use the general form of the Java integer parsing routine, `Integer.parseInt(String str, int base)` to convert such sequences to integer code.

06 (Error handling)

JavaCC throws a `TokenMgrError` when it detects a lexical error. Its error message shows useful information such as line and column numbers of the error location.

Some forms of errors that our lexer likes to report are not automatically captured by JavaCC. For instance, when an integer overflows, it is not a lexical error per JavaCC's definition if the integer literal token allows unlimited number of digits. If we use `Integer.parseInt()`, then an integer overflow will be caught by the Java compiler. However, in such case the error messages reported by Java's Exception may not contain the same set of information as those of the JavaCC lexical errors.

This program shows how to catch a Java Exception and re-wrap it as a `TokenMgrError`, to allow a more targeted error message. Note that the `TokenMgrError()` constructor takes two arguments: a message and an integer error code ("0" means lexical error).

from 06/Lexer.jj

```
        if (tkn.kind == INTLIT) {
          try {
            System.out.println(Integer.parseInt(tkn.image));
          } catch (NumberFormatException e) {
            throw new TokenMgrError("Integer overflow: " + tkn.image, 0);
          }
```

07 (File input)

This program shows how to use Java file input. For our purpose of reading characters from a file, we can use either `FileInputStream` or `FileReader`. Once an input stream is established, it works the same as `System.in`. But do remember to close a stream after use.

from 07/Lexer.jj

```
      FileInputStream stream = new FileInputStream(args[0]);
      Lexer lexer = new Lexer(stream);
```

Part 2. Manual Implementation of a Lexer

We'll walk through a sequence of programs to see some of the main issues in manual lexer implementation.

10 (A simple echo program with file input)

Shows how to handle simple file input: open a file, read a character from a file, close a file.

from 10/Lexer.java

```
        FileInputStream input = new FileInputStream(args[0]);
              int c, charCnt = 0;
              while ((c = input.read()) != -1) {        // read() returns -1 on EOF
          System.out.print((char)c);
          charCnt++;
              }
              input.close();
              System.out.println("Total chars: " + charCnt);
```

11 (A simple lexer for keyword tokens)

Shows how to use simple pattern matching technique to recognize a keyword token. In each case, the sequence of characters that form the target keyword is read one by one:

from 11/Lexer.java

```
    case 'b':
      // read the next four chars
      if ((c=input.read()) == 'e' && (c=input.read()) == 'g' &&
          (c=input.read()) == 'i' && (c=input.read()) == 'n')
        return BEGIN;
      break;
```

If a character mismatch happens, the matching process is aborted right away (credit to the short-circuit semantics of the `&&` operator). If a complete match is found at the end, the keyword is recognized.

Note that this is a naive approach for handling keyword tokens, and it does not generalize well. The preferred approach in real lexers is to use a buffer to save the lexeme string first, and then to post-process it for keyword recognition.

12 (A lexer for ID tokens)

This program shows the use of an input buffer for helping recognize ID tokens. There are two reasons for using a buffer. One, the lexeme to an ID token is an essential piece of the token's information; and two, when keyword tokens are present, the buffer will facilitate the post-processing for distinguishing keywords from IDs.

In many programming languages, including Java, identifiers do not have a length limit. As such, we cannot use a fixed-size buffer to hold an identifier's lexeme. In this program, we use an `StringBuilder` object for this purpose. It can hold a string of arbitrary size and it allows new characters to be appended at the end. (The class `StringBuffer` has the same properties, but is slightly more expensive.)

from 12/Lexer.java

```
static int nextToken() throws Exception {
  StringBuilder buffer = new StringBuilder();
  ...
      if (isLetter(c)) {
        // save ID token's lexeme
        buffer.setLength(0);
        do {
          buffer.append((char) c);
          c = input.read();
        } while (isLetter(c));
        lexeme = buffer.toString();
        return ID;
```

13 (Version 2, Token objects)

This program is a more sophisticated version of the previous one. It defines a class for representing tokens, encapsulating all relevant information, such as token code, lexeme, line and column numbers, inside a class object.

The `Token` class is defined as an *static inner class* inside the lexer class. This is a programming style choice. It allows related program parts co-exist in the same file for easier access. When compiled, each class still has a separate `.class` file. (The alternative is to define each class in a separate file to begin with.)

from 13/Lexer.java

```
// Token object
static class Token {
  int kind;           // token code
  int line;           // line number of token's first char
  int column;         // column number of token's first char
  String lexeme;      // lexeme string
  ...
}
```

This program also shows how to book-keep line and column numbers. To do this, we need to track each new character input, and track the newline characters. It is convenient to define a separate function `nextChar()` to wrap the `read()` function with the book-keeping activities.

from 13/Lexer.java

```
static int nextChar() throws Exception {
  int c = input.read();
  if (c == '\n') {
    linNum++;
    colNum = 0;
  } else {
    colNum++;
  }
  return c;
}
```

Exercise: Add two keyword tokens, `begin` and `end`, into this lexer.

14 (Version 3, Error handling)

In this version, we add in the error handling feature. While the pre-defined Java Exception class can provide the service, we would like to have a more customized version for our lexer. For this purpose, we use a *static inner class* again:

from 14/Lexer.java

```
static class LexError extends Exception {
  int line;
  int column;
  public LexError(int line, int column, String msg) {
    super("at line " + line + " column " + column + ": " + msg);
    this.line = line; this.column = column;
  }
}
```

This version of lexer skips only white space characters, and reports error for other non-letter characters, so there are changes in the the routine `nextToken()` as well.

from 14/Lexer.java

```
      if (isLetter(c)) {
        ...
        return new Token(ID, beginLine, beginColumn, buffer.toString());
      }
      throw new LexError(linNum, colNum, "Illegal char: " + (char)c);
```

Exercise: Add operator tokens, `+`, `-`, `*`, and `/`, into this lexer.