CS164: Programming Assignment 2
Dlex Lexer Generator and *Decaf* Lexer

Assigned: Thursday, September 16, 2004
Due: Tuesday, September 28, 2004, at **11:59pm**

September 16, 2004

1 Introduction

Overview

In this assignment, you will develop *Dlex*, a lexer generator nearly as powerful as the well-known `lex`. Next, you will *Dlex* to develop a lexer for *Decaf*, our classroom subset of Java.

The purpose of this project is for you to (1) observe that relatively complicated programs (in our case, the lexer) can be generated automatically from a short high-level specification; (2) learn how to write such a code generator (in our case, a lexer generator); and (3) use your lexer generator to produce the lexer for your *Decaf* compiler.

Also, you will get practice with how to use CVS for collaboration with your partner. See the web site (Course Info) for details.

On this project, you will work in pairs. Please form your team as soon as possible.

What you are going to learn and why it is useful

First, you are going to learn how to use finite automata in practice. Specifically, you are going to see that after extending a standard non-deterministic finite automaton (NFA) with a simple lookahead capability, you can use the NFA to break a character string into tokens (more accurately, into lexemes). Such a "tokenizer" is useful not only in a compiler's lexer, but also any time you need to process some text, whether it is a program or a data file.

Second, you will learn that ASTs can represent not only programs in what you may think of as "standard" programming languages like Java or *Decaf*, but also programs that are really high-level specifications, such as a description of what a lexer is supposed to do. In our case, you are going to learn that regular expressions can be represented as ASTs. We are going to use these regular expression ASTs to translate regular expressions into (i) NFAs and (ii) a code that, when executed, constructs the NFA.

Furthermore, you will see that the choice of the syntax in which the program is expressed is not

terribly important. In particular, we will express regular expressions not in their usual syntax (using operators '.', '|', and '*') but instead in the Java syntax. This will allow us to create ASTs of regular expressions with the Eclipse Java parser you used in PA1. The cost of using Java syntax is minor clumsiness in writing regular expressions, but that's worth it compared to the alternative of having to create a new parser specifically designed for regular expressions.

Third, you are going to use a more interesting way of interpreting a program than you played with in PA1. Specifically, you are going to traverse an AST of a regular expression (not that of a *Decaf* program) and your evaluation of the AST will construct the corresponding NFA (as opposed to executing the AST program).

Finally, you are going to learn how to generate code. Code generation will complete your task of translating a high-level specification (regular expressions) into a (Java) code that performs the specified task (it constructs the lexer). Note that you already had done code generation in cs164: The pretty-printer in PA1 took a *Decaf* AST as input and printed the program as *Decaf* source text. The pretty-printer can be viewed as a code generator that emitted (nearly legal Java) code, which when executed on a Java VM would execute the AST program.

2 The Assignment

Your goal in PA2 is to

1. implement a lexer generator called *Dlex*,
2. use *Dlex* to generate a lexer for *Decaf*, and
3. develop a sufficient collection of test cases to test your lexer generator and your *Decaf* lexer.

Your lexer generator must accept the token specification given in Section 2.3. Your lexer must partition the input into tokens according to the *Decaf* language definition in Section 2.1. Also, the lexer must follow the guidelines in Section 2.2.

Note: You may want to skip the rest of Section 2 on the first reading. Although this section contains important details, it is sufficient to refer back to these details as you are progressing through the steps of the assignment described in Section 3.

2.1 *Decaf*'s Language Definition (Lexical Specification)

The tokens of the *Decaf* language are defined as follows:

- Any of the following lexemes are **keywords**:

public	class	static	int	void	extends	print	new	this
true	false	if	else	while	return	boolean	null	

- **Identifiers**. An identifier starts with an underscore or letter and is followed by zero or more letters, digits or underscores. Reserved keywords (those listed above) are not identifiers.

- **Integer literals**. An integer literal is a sequence of one or more decimal digits.
- Any of the following one- or two-character **symbols**:

```
{       }       (       )       ,       =       ;       .

+       -       *       /       !       &&      ||      ==

!=      <       >       <=      >=      <<      >>
```

- **String literals**. A string literal is a sequence of zero or more *string characters* surrounded by double quotes. A string character is one of the following:
  - a single ASCII character other than
    - a control character (a control character is any character in the range ASCII 0-31, which includes newline and carriage return),
    - double quote (ASCII 34),
    - backslash (ASCII 92),
    - delete (ASCII 127).
  - a two-character escape sequence, which is a backslash followed by a single quote, double quote, backslash, n, or space.

  Extended ASCII characters (with ASCII codes greater than 127) cannot appear in string literals. The newline character may appear as `'\n'`, `'\r'`, or `'\r' + '\n'`.

  Examples of legal string literals:

```
""

"&!#"

"use \n to denote a newline character"

"include a quote like this \" and a backslash like this \\"
```

  Examples of things that are not legal string literals:

```
"unterminated

"also unterminated \"

"bad escaped character: \a AND not
   terminated
```

- **Comments.** Comments in *Decaf* are like Java single line comments: they start with `//`, include any printable ASCII character (values 32-126), and continue until end-of-line. They should not contain tabs. The comment must not end with an EOF. `UnmatchedException` should be raised in the lexer if the comment contains tabs or it ends with an EOF.

Note that the lexer should recognize comment lexemes, but that comments are not tokens. Comment lexemes are to be ignored and no token should be returned to the parser. To ignore a token, the token id should be `TokenMap.IGNORE` to have the `LookaheadLexer` ignore the token. This means you should use `return IGNORE;` in your lexer specification. (See part 1 on the `LookaheadLexer` for more details).

- **Whitespace.** Spaces, tabs, and newline characters are whitespace. Whitespace separates tokens, but should otherwise be ignored (except inside a string literal).
- **Illegal characters.** Any character that is not whitespace and is not part of a token or comment is illegal. For any string or character that is not matched, an `UnmatchedException` should be raised in the lexer. One example is `"unterminated`, which is an unterminated string.

2.2 Lexer specification

Your lexer must behave according to the following rules:

- Token types are defined in the file `tests/decaf.dtok`. For example, the name for the token to be returned when an integer literal lexeme is recognized is `INTLITERAL`. Please see `tests/decaf.dtok` for a complete list.
- You may not assume any limits on the lengths of identifiers, string literals, integer literals, comments, etc.

2.3 Specification of the Lexer Generator *Dlex*

Each lexer description consists of one `.dlex` file and one `.dtok` file. The `.dlex` file specifies the regular expressions that form the tokens of the language. The `.dtok` file specifies a mapping between the symbolic token names and the integer constants that represent token types. We will refer to these two files as the lexer specification and token map, respectively. Both of these files will be in Java syntax, so that we can parse them with the Eclipse Java parser (see the `util` package in the starter kit).

See `tests/TestNfa.dlex` and `tests/TestNfa.dtok` for an example of a lexer specification and its associated token map. These two files are also shown here:

Sample Lexer Spec (see `tests/TestNfa.dlex`)

```
public class Lexer {
    public static void main() {
        digit = '0' - '9';
        alpha = 'a' - 'z' | 'A' - 'Z';

        if( "while" ) { return WHILE; }
        if( digit + digit + digit ) { return AREACODE; }
        if( digit*1 ) { return NUMBER; }
        if( alpha + (digit | alpha | '_')*0 ) { return ID; }
```

```
    }
}
```

Sample Token Map (see `tests/TestNfa.dtok`)

```
public class Tokens {
    public static void main() {
        WHILE    = 1;
        NUMBER   = 2;
        ID       = 3;
        AREACODE = 4;
    }
}
```

Lexer Spec Format:

- The body of the lexer spec is divided in macros and regular expression patterns.
- Macros are used for defining commonly used patterns, such as the set of letters. The macro can then be used in constructing more complicated regular expressions. A macro is defined by using the assignment operator. For example, `digit = '0' - '9';`, assigns the name "digit" to be the range from '0' to '9'. The following macro definition is also very useful: `alpha = 'a'-'z' | 'A' - 'Z'`.

  Macros in .dlex files may be defined in terms of other macros. You can assume that a macro will be defined before it is used in another definition (i.e., we will never feed your generator with a .dlex file where this is not the case)..
- A regular expression specification is defined by an `if` statement. The regular expression to match is written in the boolean expression. The type of token is specified by the return statement in the "then" block of the `if` statement. For this part, you can assume there will be exactly one return statement, and no other statements, in the body of an if. For the range regular expression operator -, you can assume that both arguments are character literals.
- In general, no error checking is necessary in *Dlex* (e.g., checking for uses of undefined macros), but feel free to implement some error checking to help debug your .dlex files.
- Regular expressions can consist of the following:

`'a'`	Matches a single character.
`'a'-'z'`	Matches a range of characters with ASCII values between the first and second character, in
`"string"`	Matches a non-empty string of characters.
`re1 | re2`	`|` is the or operator. Matches either `re1` or `re2`.
`re1 + re2`	`+` is the concatenation operator. Matches `re1` followed by `re2`.
`re *0`	`*0` is the star operator. Matches zero or more instances of `re`.
`re *1`	`*1` is the plus operator. Matches one or more instances of `re`.
`macro`	Matches the regular expression defined by the macro.

- The order of token specifications matters. The tokens defined earlier should have higher precedence than those defined later.

Token Map format:

- The token map ".dtok" file maps symbolic name of tokens to integers. For example, the above token map file maps `WHILE` to the integer 1, which is what will be returned to the parser. In the lexer, the type of the token will be represented by the number 1, rather than the string "WHILE".
- The body of a token map consists of a sequence of assignments from the token symbolic name to the integer for that type.
- Type numbers ≤ 0 are reserved for special tokens. You may use numbers ≥ 1 for token types.

3 Implementation notes

3.1 The starter kit

The starter kit is available on the web at:

`http://www-inst.eecs.berkeley.edu/~cs164/starters/PA2.zip`

You can also get the starter kit from your Unix account at:

`~cs164/public_html/starters/PA2.zip`.

Usage of the starter kit is the same as PA1. Please to refer to the PA1 handout for information on how to import the zip file into a new project.

3.2 Project Steps

You will be building your lexer and lexer generator one piece at a time, which will help you verify the correctness of your code at each step. This project is divided into 5 steps.

The harness code that is used to run each of the steps are contained in `Main.java`. They are functions `runStep0()`,..., `runStep4()`. The harness code given will show you how to set up the different parts of the code for testing, however remember when grading, we will be running your code on many more test cases than what is given in `Main.java`.

In general for this project, feel free to add fields or methods to any of the classes that we have provided. However, make sure that the methods that the `Main` class relies upon are not broken, as they may be used in remote testing or grading.

3.3 Remote Testing

To make sure that we don't overwhelm you with a complex remote-testing setup, PA2's remote testing server will test only the final solution (i.e., the code you will write in Step 4). Remote testing won't be not available for the intermediate steps. These steps are easier than Step 4, but you should still test these steps carefully.

When remote-testing your lexer generator, we will provide our own `Main.java` file.

3.4 Grading

When grading your solution, we will test both your lexer generator and your *Decaf* lexer. That is, we will feed your lexer generator a spectrum of lexer specifications and we'll test the lexers your generator produces from these specifications.

3.5 Do I need to follow the six steps outlined in this handout?

No. As long as your lexer generator and the *Decaf* lexer it produces behave according to the specification, you can design your generator and lexer as you find suitable. Even if you decided not to follow the steps we suggested, read the directions given under each step carefully: they contain requirements that your solution will need to meet.

Step 0: Recognizer

You will not need to code anything for step 0. The starter kit for PA2 contains an implementation of a `Recognizer`. A `Recognizer` takes as input an NFA and a string, and determines whether or not the NFA accepts the string. The sample code `runStep0()` in `Main.java` creates an NFA and runs the recognizer on several sample strings. Understanding the code of `Recognizer` is necessary for developing the `LookaheadLexer` in Step 1.

Classes to Modify: You will not need to modify any code for step 0.

Other Classes Needed:

`Recognizer`	Code for the Recognizer.
`SimpleNfa`	Creates the NFA that can be used for testing of the recognizer.
`NFAState`	Represents the NFA in a graphical fashion, with nodes and transitions.
`Main`	Test harness code (for your local testing, not remote testing).

Step 1: LookaheadLexer

In Step 1, you will implement a `LookaheadLexer`. `LookaheadLexer` takes as input a file stream and an NFA, and uses the maximal munch rule to recognize tokens from the file.

Requirements:

- You must implement the code for `nextToken()` in `LookaheadLexer`. `nextToken()` will return the next token read from the file stream. This is the method that will be called by the parser.
- At the end of the stream, `Token.EOF` must be returned to signal the end of the file.
- `LookaheadLexer` must use the maximal munch rule. For example, in `SimpleNfa`, there is a token for "while" and identifiers, which are strings of letters. Given the string "whileandmorecharacters", `nextToken()` must return the full string, rather than a "while" token. You will need to implement a mechanism which will remember the last final state that the NFA was in, and return the token at that state when the NFA becomes stuck.

  The basic algorithm for maximal munch is:

```
while(!nfa is stuck) {
    make a transition on next character and epsilon moves
    if(one of the states is final) {
        remember the state with the highest priority and the current
        position in the input stream
    }
}
// now the nfa is stuck
1. push input characters since the last final state back
   onto the stream
2. return the token which represented at the last final state
```

- If the input does not match a token, then an `UnmatchedException` must be raised.
- If the token type of the final state in the NFA is `TokenMap.IGNORE`, then your `LookaheadLexer` should discard that token, and return the next non-IGNORE token in the input.
- If the NFA stops in multiple final states, you must return the token that has the largest priority.

Testing:

You can use the NFA created by `SimpleNfa` to test your `LookaheadLexer`. The tokens recognized by `SimpleNfa` are:

`while`	The keyword "while"
`identifiers`	Strings of letters, 1 character and longer
`positive integers`	Strings of digits, 1 digit and longer

The "while" token has higher priority than identifiers.

Classes to modify:

`LookaheadLexer`	You will need to implement `nextToken()` to match the requirements above.

Other classes needed:

`NFAState`	Represents the NFA in a graphical fashion, with nodes and transitions.
`SimpleNfa`	Creates the NFA that can be used for testing.
`UnmatchedException`	Exception thrown by the lexer when input no tokens.
`Main`	Test harness code.

Step 2: LexerNfaGenerator

In part 2 of this assignment, you will implement the visitor class that converts an AST for a lexer specification to an NFA data structure. You will first use `Parser` to generate the AST from a ".dlex" lexer specification file, and then use the `LexerNfaGenerator` to build the NFA.

Requirements:

- `LexerNfaGenerator` will perform NFA construction from the AST parsed from a ".dlex" file. You will need to write the `visitor(...)` methods and implement the algorithm for constructing an NFA from a regular expression's AST. This NFA should look like the one in `SimpleNFA`.
- Token types are stored in the `NFAState` as integer constants. However, they are stored in the AST as `String`'s. You will use a `TokenMap` to map symbolic token names to integers. `TokenMap` is produced by `TokenMapGenerator`, which reads in a seperate ".dtok" file containing the mappings. `TokenMapGenerator` is written for you in the starter kit.
- You will need to extract the symbolic token name out of the return statement.
- Be sure you handle the precedence of the rules correctly. Remember, rules listed earlier have higher precedence than rules listed later in the lexer specification. When you are constructing the NFA, you must give the final states for regular expressions that came earlier in the specification a higher priority number. Also, when making NFA states final, use the `DUMMY` action from `LexerAction` as the action parameter; we'll discuss actions more in Part 4.

Testing:

You should modify `tests/TestNfa.dlex` and `tests/TestNfa.dtok` to test the ".dlex" specification constructs.

Classes to modify:

`LexerNfaGenerator`	You will need to implement the `visit(...)` methods in this class.

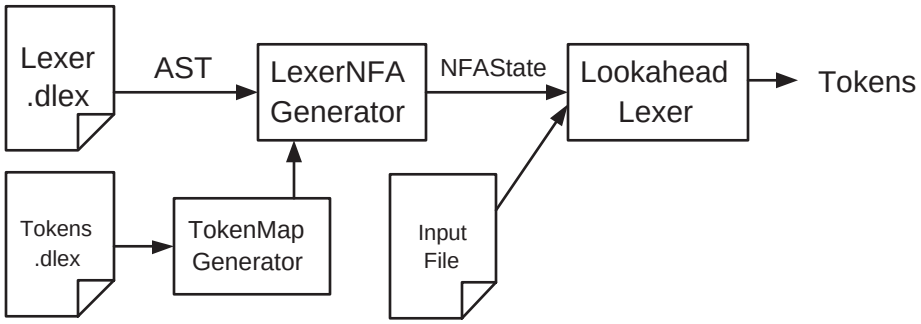
Other classes needed:

`TokenMap`	Maps tokens to token type id's.
`TokenMapGenerator`	Generates `TokenMap`'s.
`Parser`	Parses ".dlex" file to an AST
`NFAState`	Represents the NFA in a graphical fashion, with nodes and transitions.
`Main`	Test harness code.

Step 3: Decaf Token Specification

In this part of the project, you will write the token specification for the *Decaf* language. By now, you have written the `LexerNfaGenerator` and the `LookaheadLexer`, and so you will be able to use them to convert a *Decaf* input program into tokens.

Your lexer setup will be as shown below. Note that your lexer generator as of this step will generate an NFA (i.e., a data structure) rather than a lexer code (i.e., the lexer that will produce an NFA). The latter will be done in the next step.

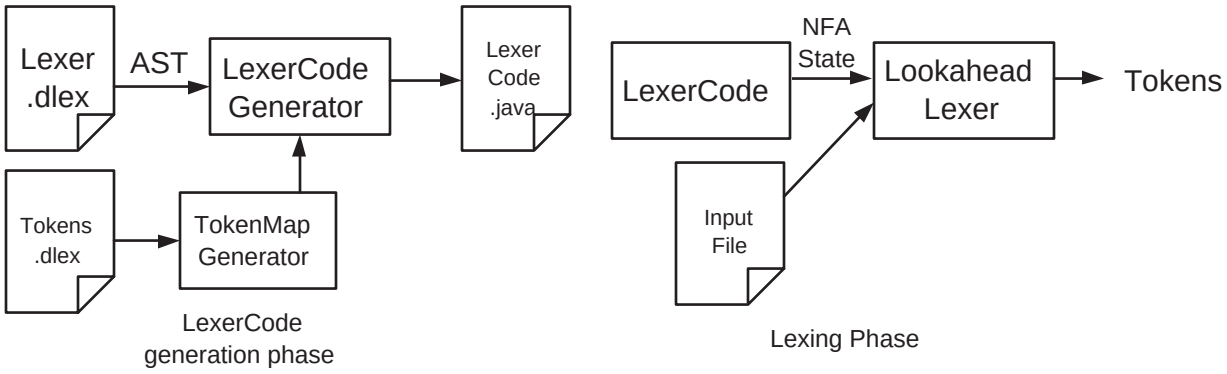


Requirements:

- You should write the lexer specification file in `tests/decaf.dlex`. The token map for the symbolic token names has already been provided to you in `tests/decaf.dtok`.
- Test files you use for testing the lexer should be placed in the `test/` directory.

Classes to modify: None.

Step 4: LexerCodeGenerator and LexerCode



In the final phase of this project, you will implement the `LexerCodeGenerator`. In the previous steps of the project, you used `LexerNfaGenerator` to directly create the NFA data structure used by the `LookaheadLexer`. What the `LexerCodeGenerator` will do instead is generate a Java file `LexerCode.java` containing a class `LexerCode`. The `LexerCode` class will contain a method `getNFA()` which will be hardcoded to generate the NFA. In fact, `SimpleNfa` used in step 1 is an example of a piece of NFA-generating code that was produced by a `LexerCodeGenerator`.

Below is a sample of the input and output of `LexerCodeGenerator`. Our lexer in this case matches only `JUSTAPLAINA` token, with id = 1, consisting of just the letter 'a'. Note that this is just an example of how `LexerCodeGenerator` could work; your generated code does not have to look similar to the example code.

Lexer specification file

```
public class Lexer {
    public static void main() {
        if('a') { return JUSTAPLAINA; }
    }
}
```

Lexer token map file

```
public class Tokens {
    public static void main() {
        JUSTAPLAINA = 1;
    }
}
```

`LexerCodeGenerator` output

```
public class LexerCode {
    // generated function that returns the NFA for the input
    // specification file
    public NFAState getNFA() {
        NFAState start, end;
```

```
        // create code corresponding to matching JUSTAPLAINA token
        start = new NFAState();
        end   = new NFAState();

        start.addTransition('a', end);
        end.makeFinal(1, 1, null);

        // return NFA
        return start;
    }
}
```

Lexer actions

A lexer *action* is an arbitrary piece of Java code that is run whenever a particular token is recognized. For instance, suppose we want to add the functionality to count the number of JUSTAPLAINA tokens that were in the source file. One way to do this is to add a piece of code that increments a counter whenever we find a match for a JUSTAPLAINA token. The action associated with a final state should be executed only when the corresponding token is about to be returned, not whenever the final state is reached.

In order to support this functionality we will add the following to our lexer specification language:

- Code statements that come in the "then" clause of the if statement before the return statement. These statements will be executed whenever the regular expression is matched.
- We won't be supporting arbitrary Java code, just a subset of the language you implemented an interpreter for in pa1. This language will have the following statements:

`name = expression ;`	assignments to variables
`print(expression) ;`	print function
`+, -, *, /`	simple arithmetic expressions
`0, 1, ...`	integer numbers

Below is the example, extended with our action:

Lexer specification file

```
public class Lexer {
    public static void main() {
        if('a') {
            numPlainA = numPlainA + 1;

            return JUSTAPLAINA;
        }
    }
}
```

`LexerCodeGenerator` output

```
public class LexerCode {
    // note that we generated a static variable in the LexerCode
    // class for each variable that our action code uses.
    public static int numPlainA = 0;

    // generated function that returns the NFA for the input
    // specification file
    public NFAState getNFA() {
        NFAState start, end;

        // set up NFA for JUSTAPLAINA token
        start = new NFAState();
        end   = new NFAState();

        start.addTransition('a', end);

        // note that we also now register a new Action with
        // the final state of JUSTAPLAINA token
        end.makeFinal(1, 1, new Action1());

        // return NFA
        return start;
    }

    // generate a new Action class for each different token that has an
    // action associated with it. The new action class must implement
    // the interface LexerAction, which contains the method run().
    class Action1 implements LexerAction {
        // Inside the run method we will place all of the code
        // associated with the action. In this case, we have the
        // single statement which increments the count
        public void run() {
            numPlainA = numPlainA + 1;
        }
    }
}
```

What you need to do:

The basic strategy for writing `LexerCodeGenerator` is to (i) understand what code `LexerNfaGenerator` executes when constructing an NFA, and then (ii) print this code into the body of a method, which can then be called to create the NFA. Essentially, writing the code generator essentially boils down to taking the code you wrote in `LexerNfaGenerator` and printing this coed out using `print()` methods. Note: you are not supposed to create an AST for the lexer and then pretty print the AST; instead, print the lexer code directly, without first building the AST; it's simpler this way.

However, there are some subtleties in how this is to be done:

- You will need to print the class declaration and method declaration to the file:

```
public class LexerCode implements LexerCodeInterface {
    ...
    public NFAState getNFA() {
        ...
    }
}
```

- For every token, you will need to output code for constructing the NFA for the token (i.e. similar code to the code used by `LexerNfaGenerator`).
- The NFA building code requires fresh names for intermediate variables. You must ensure that the variables name you used to construct the NFA are unique. For example, if your code for the star operator is (`start` and `end` are the old start and end states):

```
NFAState newStart = new NFAState(), newEnd = new NFAState();

newStart.addTransition(NFAState.EPSILON, start);
newStart.addTransition(NFAState.EPSILON, newEnd);
end.addTransition(NFAState.EPSILON, newStart);

start = newStart; end = newEnd;
```

  The variables newStart and newEnd will generate a name conflict if you needed to generate the NFA for a star operator multiple times. This is because by simply reprinting the code segment, you will have multiple, conflicting definitions for the `newStart` and `newEnd` variables.

- For every token that has action statements associated with it, you need to:
  - Create an inner class in `LexerCode` that implements the `LexerAction` interface. In the above example, this is the `class Action1 { ...}` declaration. If you prefer, you can instead create anonymous classes inline with your other code.
  - Fill in the body of the run method. You should use the provided `ActionCodeGenerator` visitor class to print out the action statements to the output file.
  - Output code that will add an instance of the inner class to the final state of the NFAState. In the above code segment, this emitting code to call `NFAState.makeFinal` to link a new instance of the action class to the final state.
  - Actions in your lexer will be very simple. All of the variables will be integers, and they will all be initialized to 0.
  - For every variable that is referenced in the action code, you will need to create a `static int` variable in the `LexerCode` class. `ActionCodeGenerator` contains a method `getNameSet()` which returns a set of all of the variables that were referenced in the action code.

Requirements:

- You will implement the `visit()` methods and `emitCode()` methods in `LexerCodeGenerator`.
- After you implement `LexerCodeGenerator`, you can test it by using it to generate `LexerCode.java` using your *Decaf* specification. Then you can use `Main.runStep4()` and the test files from step 3 to test your code.
- You will also need add code to `nextToken()` in the `LookaheadLexer` to call the `run()` method in the action class associated with the final states of the NFA.
- You will modify the `Token` class to add the line number and the char number to each token object. You need to implement this by adding actions. Modify the `toString` method to print out the line number and the char number along with the the lexeme and the token type.

Classes to modify:

`LexerCodeGenerator`	You will need to implement the `visit()` and `emitCode()`.
`LookaheadLexer`	You will need to add code to `nextToken()` to implement the actions.
`LexerCode`	You should emit code to `LexerCode` in order to test your code generator.

Other classes needed:

`TokenMap`	Maps tokens to token type id's.
`TokenMapGenerator`	Generates `TokenMaps`.
`SimpleDecafParser`	Parses ".dlex" file to AST
`ActionCodeGenerator`	Pretty-prints the action code for you
`NFAState`	Represents the NFA in a graphical fashion, with nodes and transitions.
`Main`	Test harness code

4 Relevant Reading

You may wish to skim these chapters in the textbooks, as they cover lexical analysis and lexer generators. The books are on reserve in the engineering library.

Chapter 3 from the Dragon Book covers lexical analysis. Section 3.3 describes regular expressions in great detail, and Section 3.5 gives an example of how to write a lexical specification for the lexer generator `lex`.

Chapter 3 from Crafting a Compiler also has detailed coverage on the lexer phase of the compiler. Section 3.2 has coverage of regular expressions, Section 3.6 has the rules for transforming regular expression to finite automata.

The other two textbooks explain lexing in detail, too.

Handing in the Assignment

We will be collecting your assignment from your CVS repository, as in PA1. See PA1 submissions instructions for details, but check the course web page for updates before making the submission.

Good luck!