

Homework 1: Lexical Analysis

Out Date: 09/06/2007
Due Date: 09/13/2007

Objectives

The purpose of this homework is to (1) setup your development environment with eclipse, (2) get familiar with the lexical analysis development environment we use (JFlex) (3) review your skills at defining regular expressions, and (4) get you started on the compiler p4oject of the semester!

Handout

To get started on the homework you need to download the archive `hw1-package.zip` from the website of the class. This archive is a template project that you can import into Eclipse. It contains some support code and basic structure. It is up to you to “fill in” the blanks. It also contains an `rtf` file where you will list your regular expressions and some explanations for the most complex ones. Once you are ready to handin the homework, archive the directory again (as a zip file) and submit it as an attachment to the homework submission e-mail address:

`c244fa07@cse.uconn.edu`

Submission is due by MIDNIGHT on the due date.

Regular Expressions and Automata

Your first task is to write the regular expression that will recognize any lexeme that belongs to a token class of C-. For reference, C- has the following token classes:

Separator ‘;’ is used to separate statements

Binary operators C- uses the following set `+, -, *, /, <=, >=, ==, !=, <, >, &&, ||, !`

Brackets C- uses brackets, parenthesis and curly braces and pairs of brackets (for array specifications)

Dots C- uses a single dot ‘.’ for method invocation and a double dot for array size specification (as in `int [] tab = new int[1..10];`).

Comma C- separates arguments in method invocation with a comma. ‘,’.

Keywords C- uses the following keywords `int, bool, void, while, if, else, for, self, class, extends, new, return`.

Integers C- represent integers in decimal. An integer is any sequence of digit that either does not start with zero, or, if it starts with zero, it contains only 1 digit (the zero itself). For instance, `123,45,0` are valid integers but `0124` is not.

Identifiers The C- identifiers start with an letter in the roman alphabet or, possibly, an underscore. The remainder of an identifier can contain any number of alpha-numeric symbol including an underscore. For instance `hello,id3,i,k.25` are all valid identifiers but `hello-you, home%5` are not.

Whitespaces Whitespaces are used for clarity and carry no meaning at all. Whitespaces include any number of blank, tabulation, carriage return or line feed.

Comments C- support C style comments. A comment starts with `/*` and ends with `*/` with anything in between except a comment ending sequence (`*/`). A comment can span multiple lines of text. For instance `/* hello */`, `/****** hello *****/`, `/* a new test ** / ** ends here */` are all valid comments.

For each token class, you are expected to produce a regular expression that recognizes any lexeme that belongs to that class. For keywords, feel free to define one token class per keyword (this will be convenient later on).

Important. The only allowed operators that you can use for regular expressions are `?`, `+`, `*`, `|` and concatenation.

To define symbols you may also use ranges `-` and negations `[^]`.

Setting your working environment.

To get started with the development for the class perform the following steps:

- 1) install Java JDK from <http://java.sun.com/javase/downloads/index.jsp> (if you don't have it already)
- 2) install eclipse (if you don't have it already)
- 3) download JavaCUP 11a beta sourcecode release from <http://www2.cs.tum.edu/projects/cup/>
unzip into a directory
- 4) Start eclipse
go to preferences -> Java -> Build Path -> User Libraries
select New..
enter JavaCUP
select Add JARs
enter java-cup-11.jar from your file system.
enter JFlex.jar from your file system.
- 5) go to preferences -> Ant -> Runtime -> Classpath
select Global entries
select Add external JARs
enter JFlex.jar from your file system
enter java-cup-11.jar from your file system
select Tasks
select Add Task
enter JFlex
select the JFlex.jar in the Location box
select /JFlex/anttask/JFlexTask.class and click OK
select Add Task
enter CUP
select the java-cup-11.jar in the Location box
select /java_cup/anttask/CUPTask.class and click OK

JFlex

Now that you have specified the patterns for your token, your next task is to create a `lexer.flex` file in the Java template we provide. This file will be used by JFlex generate to a Java-based scanner. You are strongly encouraged to read the JFlex documentation. Your patterns should be based *purely and exclusively* on the regular expressions you defined above.

For building your homework, an "Ant build file" (`build.xml`) is included that will automatically invoke JFlex to generate your scanner from eclipse (simply import the `build.xml` file into the ant view).

Driver for the lexer

Your last task is to write in the `Driver.java` a driver class that will repeatedly call the `nextToken` method of the scanner to obtain the stream of tokens and prints it back out on the standard output. The scanner itself should be discarding the worthless tokens (white spaces and comments) and therefore should not return such tokens to the parser driver. Your driver class should produce output that looks like that :

Sample Input / Output

This is an example of C- file `test.cmm` :

```
class Foo {
int fact(int n) {
return 0;
}
int fib(int x) {
return 1;
} /* /* comment ****/
};

class Main extends Foo {
Main() {
int x;
x = fact(5);
}
int fact(int n) {
if (n==0)
return 1;
else return n * fact(n-1);
} /* ***** / *another comment*/
};
```

This is the output of the scanner :

```
scanning [test.cmm]
Token: class
Token: id = Foo
Token: {
Token: int
Token: id = fact
Token: (
Token: int
Token: id = n
Token: )
Token: {
Token: return
Token: number = 0
Token: ;
Token: }
Token: int
Token: id = fib
Token: (
Token: int
Token: id = x
Token: )
Token: {
Token: return
Token: number = 1
Token: ;
Token: }
```

```
Token: }
Token: ;
Token: class
Token: id = Main
Token: extends
Token: id = Foo
Token: {
Token: id = Main
Token: (
Token: )
Token: {
Token: int
Token: id = x
Token: ;
Token: id = x
Token: :=
Token: id = fact
Token: (
Token: number = 5
Token: )
Token: ;
Token: }
Token: int
Token: id = fact
Token: (
Token: int
Token: id = n
Token: )
Token: {
Token: if
Token: (
Token: id = n
Token: ==
Token: number = 0
Token: )
Token: return
Token: number = 1
Token: ;
Token: else
Token: return
Token: id = n
Token: *
Token: id = fact
Token: (
Token: id = n
Token: -
Token: number = 1
Token: )
Token: ;
Token: }
Token: ;
Token: eof
```

Have fun!