CS4413 WA Compilers Design

Dr. Sabah Mohammed

Department of Computer Science

Lakehead University

Exercise 1: C-- Lexical Analysis Using JFlex

Your 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 [ˆ].

1. Develop the Jflex Specification

Now that you have specified the patterns for your token, your task is to create a **lexer.flex** file in the Java template that is provided on your WebCT. This file will be used by JFlex to generate 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.

1. Write the Driver for the lexer

Your next task is to write the **Driver.java** a driver class that will repeatedly call the nextToken method of the scanner/lexer 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. You may use the **Toke.java** class (provided) at your driver. Your driver class should produce output that looks like the provided example below:

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: ;**

**Token: eof**

Have fun!