Coursework 2

This coursework is worth 10% and is due on 2nd January at 16:00. You are asked to implement the Sulzmann & Lu lexer for the WHILE language. You can do the implementation in any programming language you like, but you need to submit the source code with which you answered the questions, otherwise a mark of 0% will be awarded. If you use Scala in your code, a good place to start is the file `lexer.sc` and `token.sc` uploaded to KEATS. The template file on Github is called `cw02.sc`. The example files are in the subdirectory `examples`. The main function that will be tested is called `tokenise`. The marks will be distributed such that 3 marks are given for the correct `WHILE_REGS` regular expression; 5 marks for the correct `inj` and `mkeps` definitions; and two marks when `tokenise` produces the correct results for the example files.

⚠ Testing

For the marking, the functions that will be tested are `tokenise`, `inj` and `mkeps`.

⚠ Disclaimer

It should be understood that the work you submit represents your own effort. You have not copied from anyone else including CoPilot, ChatGPT & Co. An exception is the Scala code from KEATS and the code I showed during the lectures, which you can both freely use. You can also use your own code from the CW 1.

Question 1

To implement a lexer for the WHILE language, you first need to design the appropriate regular expressions for the following eleven syntactic entities:

1. keywords are

   while, if, then, else, do, for, upto, true, false, read, write, skip

2. operators are: +, -, *, %, /, ==, !=, >, <, <=, >=, :=, &&, ||

3. letters are uppercase and lowercase

4. symbols are letters plus the characters ., _, >, <, =, ;, , (comma), \ and :

5. parentheses are (, {,) and }

6. digits are 0 to 9

7. there are semicolons ;

8. whitespaces are either " " (one or more) or \n or \t or \r

9. identifiers are letters followed by underscores _, letters or digits

10. numbers for numbers give a regular expression that can recognise 0, but not numbers with leading zeroes, such as 001

11. strings are enclosed by double quotes, like "...", and consisting of symbols, digits, parentheses, whitespaces and \n (note the latter is not the escaped version but \ followed by n, otherwise we would not be able to indicate in our strings when to write a newline).

12. comments start with // and contain symbols, spaces, parentheses and digits until the end-of-the-line markers

13. endo-of-line-markers are \n and \r\n

You can use the basic regular expressions

$$\mathbf{0},\ \mathbf{1},\ c,\ r_1 + r_2,\ r_1 \cdot r_2,\ r^*$$

but also the following extended regular expressions

$[c_1, c_2, \ldots, c_n]$	a set of characters
r^+	one or more times r
$r^?$	optional r
$r^{\{n\}}$	n-times r

Later on you will also need the record regular expression:

$$REC(x:r) \quad \text{record regular expression}$$

Try to design your regular expressions to be as small as possible. For example you should use character sets for identifiers and numbers. Feel free to use the general character constructor *CFUN* introduced in CW 1.

Question 2

Implement the Sulzmann & Lu lexer from the lectures. For this you need to implement the functions *nullable* and *der* (you can use your code from CW 1), as well as *mkeps* and *inj*. These functions need to be appropriately extended for the extended regular expressions from Q1. The definitions you need to create are:

$$mkeps([c_1, c_2, \ldots, c_n]) \overset{\text{def}}{=} ?$$
$$mkeps(r^+) \overset{\text{def}}{=} ?$$
$$mkeps(r^?) \overset{\text{def}}{=} ?$$
$$mkeps(r^{\{n\}}) \overset{\text{def}}{=} ?$$

$$inj\,([c_1, c_2, \ldots, c_n])\,c\,\ldots \overset{\text{def}}{=} ?$$
$$inj\,(r^+)\,c\,\ldots \overset{\text{def}}{=} ?$$
$$inj\,(r^?)\,c\,\ldots \overset{\text{def}}{=} ?$$
$$inj\,(r^{\{n\}})\,c\,\ldots \overset{\text{def}}{=} ?$$

where inj takes three arguments: a regular expression, a character and a value. Test your lexer code with at least the two small examples below:

regex:	string:
$a^{\{3\}}$	aaa
$(a + \mathbf{1})^{\{3\}}$	aa

Both strings should be successfully lexed by the respective regular expression, that means the lexer returns in both examples a value.

Also add the record regular expression from the lectures to your lexer and complete the function `env` so that it returns all assignments from a value (this then allows you to extract easily the tokens from a value in the next question).

Finally make that the function `lexing_simp` generates with the regular expression from Q1 for the string

```
"read n;"
```

the following pairs:

```
List((k,read), (w, ), (i,n), (s,;))
```

Question 3

Make sure your lexer from Q2 also simplifies regular expressions after each derivation step and rectifies the computed values after each injection. Use this lexer to tokenise the six WHILE programs in the `examples` directory. Make sure that the `tokenise` function filters out whitespaces and comments.