Coursework 2

This coursework is worth 10% and is due on 10 November 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. You need to submit your written answers as pdf—see attached questionaire. Code send as code. 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`. Your code needs to be uploaded to Github by the deadline.

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`, `to`, `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. Write down in the questionaire at the end the clauses for

$$\begin{array}{ll}
mkeps([c_1, c_2, \ldots, c_n]) & \stackrel{\text{def}}{=}\ ? \\
mkeps(r^+) & \stackrel{\text{def}}{=}\ ? \\
mkeps(r^?) & \stackrel{\text{def}}{=}\ ? \\
mkeps(r^{\{n\}}) & \stackrel{\text{def}}{=}\ ? \\
\\
inj\,([c_1, c_2, \ldots, c_n])\,c\,\ldots & \stackrel{\text{def}}{=}\ ? \\
inj\,(r^+)\,c\,\ldots & \stackrel{\text{def}}{=}\ ? \\
inj\,(r^?)\,c\,\ldots & \stackrel{\text{def}}{=}\ ? \\
inj\,(r^{\{n\}})\,c\,\ldots & \stackrel{\text{def}}{=}\ ?
\end{array}$$

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 implement a function, say `env`, that returns all assignments from a value (such that you can extract easily the tokens from a value).

Finally give **all** the tokens for your regular expressions from Q1 and the string

```
"read n;"
```

and use your `env` function to give the token sequence.

Question 3

Extend your lexer from Q2 to also simplify regular expressions after each derivation step and rectify the computed values after each injection. Use this lexer to tokenize six WHILE programs some of which are given in Figures 1 – 4. You can find these programms also on Github under the `cw2` directory. Give the tokens of these programs where whitespaces and comments are filtered out. Make sure you can tokenise **exactly** these programs.

```
write "Fib: ";
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result: ";
write minus2
```

Figure 1: Fibonacci program in the WHILE language.

```
start := 1000;
x := start;
y := start;
z := start;
while 0 < x do {
 while 0 < y do {
  while 0 < z do { z := z - 1 };
  z := start;
  y := y - 1
 };
 y := start;
 x := x - 1
}
```

Figure 2: The three-nested-loops program in the WHILE language. (Usually used for timing measurements.)

```
// Find all factors of a given input number
// by J.R. Cordy August 2005

write "Input n please";
read n;
write "The factors of n are:\n";
f := 2;
while (f < n / 2 + 1) do {
  if ((n / f) * f == n)
  then  { write(f); write "\n" }
  else { skip };
  f := f + 1
}
```

Figure 3: A program that calculates factors for numbers in the WHILE language.

```
// Collatz series
//
// needs writing of strings and numbers; comments

bnd := 1;
while bnd < 101 do {
  write bnd;
  write ": ";
  n := bnd;
  cnt := 0;

  while n > 1 do {
    write n;
    write ",";

    if n % 2 == 0
    then n := n / 2
    else n := 3 * n+1;

    cnt := cnt + 1
  };

  write " => ";
  write cnt;
  write "\n";
  bnd := bnd + 1
}
```

Figure 4: A program that calculates the Collatz series for numbers between 1 and 100.

Answers

Question 2:
(Use mathematical notation, such as r^+, rather than code, such as `PLUS(r)`)

$mkeps([c_1, c_2, \ldots, c_n]) \stackrel{\text{def}}{=}$ _____

$mkeps(r^+) \stackrel{\text{def}}{=}$ _____

$mkeps(r^?) \stackrel{\text{def}}{=}$ _____

$mkeps(r^{\{n\}}) \stackrel{\text{def}}{=}$ _____

$inj\,([c_1, c_2, \ldots, c_n])\,c\,\ldots \stackrel{\text{def}}{=}$ _____

$inj\,(r^+)\,c\,\ldots \stackrel{\text{def}}{=}$ _____

$inj\,(r^?)\,c\,\ldots \stackrel{\text{def}}{=}$ _____

$inj\,(r^{\{n\}})\,c\,\ldots \stackrel{\text{def}}{=}$ _____

Tokens for `"read n;"`

_____

_____

_____

_____