Coursework 2 (Strand 1)

This coursework is worth 5% and is due on 6 November at 16:00. You are asked to implement the Sulzmann & Lu tokeniser 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 can submit your answers in a txt-file or as pdf.

Disclaimer

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

Question 1 (marked with 1%)

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

1. keywords are

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

2. operators are

   `+`, `-`, `*`, `%`, `/`, `==`, `!=`, `>`, `<`, `:=`, `&&`, `||`

3. strings are enclosed by `"..."`
4. parentheses are `(`, `{`, `)` and `}`
5. there are semicolons `;`
6. whitespaces are either `" "` (one or more) or `\n`
7. identifiers are letters followed by underscores `_`, letters or digits
8. numbers are `0`, `1`, ...and so on; give a regular expression that can recognise `0`, but not numbers with leading zeroes, such as `001`

You can use the basic regular expressions

$$\varnothing, \epsilon, c, r_1 + r_2, r_1 \cdot r_2, r^*$$

but also the following extended regular expressions

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

Try to design your regular expressions to be as small as possible. For example you should use character ranges for identifiers and numbers.

Question 2 (marked with 3%)

Implement the Sulzmann & Lu tokeniser 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 the clauses for

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

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
$(a + \epsilon)^{\{3\}}$	aa

Both strings should be sucessfully 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 tokeniser 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 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 (marked with 1%)

Extend your tokenizer from Q2 to also simplify regular expressions after each derivation step and rectify the computed values after each injection. Use this tokenizer to tokenize the programs in Figure 1 and 2. Give the tokens of these programs where whitespaces are filtered out.

```
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.