

Compiler Construction 2016

— Solution Series 1 —



Credit: <https://xkcd.com/208/>

General Remarks

- You may hand in your solutions for the theoretical part of the exercises just before the exercise class starts *or* via email together with the practical part of the exercises (as a separately attached PDF) *or* by coming by coming to our office.
- If you have questions regarding the exercises and/or lecture, feel free to write us an email or visit us at our office.
- We will publish solutions on our website. Typically, these will be uploaded on the day you get back your corrections.
- You may use the latest official version of Java and the standard library to solve the programming tasks. Other libraries are not allowed.
- We do *not* insist on handwritten solutions.
- There really is no L2P.
- If you are still looking for a group or your group has less than 3 members, please contact us after the exercise class such that we can mediate.
- Since there are no lectures on Tuesday (25.4) and on Thursday (5.5) and there is DIES on Tuesday (3.5), there will be no exercise class next week. **The next exercise sheet is due on the 10th of May.** Note that the next sheet will also be published a bit later.

Exercise 1

(2 Points)

As an example of the language *WHILE*, consider the following program.

```
1  /* A random walk */
2  int x = 10;
3  int s = 0;
4  while ( x > 0 ) {
5      int b = read() % 2; // randomness by user input
6      if (b == 1) {
7          x = x + 1;
8      } else {
9          x = x - 1;
10     }
11     s++;
12 }
13 write("I stopped walking after: ");
14 write(s);
15 write(" steps");
```

- (a) Give a complete list of the symbol classes and corresponding tokens needed for the lexical analysis of our programming language *WHILE*. Recall that *WHILE* captures (integer) variable declarations, assignments, arithmetic operations, conditional branches, loops, basic I/O (read and write) and Java-style comments.
- (b) Decompose the if-statement (lines 6-10) into a sequence of lexemes and translate each lexeme into a symbol.

Solution: _____

- (a) There are several possible ways to solve this task. For example, all keywords may be identified with the same token and distinguished only through an attribute or each keyword may be identified by a unique token. The former solution is presented here, the latter is chosen for the implementation.

lexeme	symbol class	token	symbol
<i>int</i>	keywords	key	(key, int)
<i>read</i>	keywords	key	(key, read)
<i>write</i>	keywords	key	(key, write)
<i>while</i>	keywords	key	(key, while)
<i>if</i>	keywords	key	(key, if)
<i>else</i>	keywords	key	(key, else)
==	relation	rel	(rel, eq)
!=	relation	rel	(rel, neq)
<	relation	rel	(rel, lt)
<=	relation	rel	(rel, leq)
>	relation	rel	(rel, gt)
>=	relation	rel	(rel, geq)
!	bool. op.	bool	(bool, not)
&&	bool. op.	bool	(bool, and)
	bool. op.	bool	(bool, or)
+	arith. op.	aop	(aop, plus)
-	arith. op.	aop	(aop, minus)
*	arith. op.	aop	(aop, mult)
/	arith. op.	aop	(aop, div)
%	arith. op.	aop	(aop, mod)
++	arith. op.	aop	(aop, inc)
--	arith. op.	aop	(aop, dec)
;	special symbol	sym	(sym, semicolon)
=	special symbol	sym	(sym, assignment)
(	special symbol	sym	(sym, lpar)
)	special symbol	sym	(sym, rpar)
{	special symbol	sym	(sym, lbrace)
}	special symbol	sym	(sym, rbrace)
0 + (- + ε){1..9}{0..9}*	numbers	num	(num, value)
{a..zA..Z_}{a..zA..Z0..9_}*	identifier	id	(id, value)
"{a..zA..Z0..9_ + - * ...}*"	string constant	string	(string, value)
<i>true</i>	boolean constant	bconst	(bconst, true)
<i>false</i>	boolean constant	bconst	(bconst, false)
// ... (\n + \r + \r\n)	comment	cmt	(cmt, cmtsingl)
/ * ... */	comment	cmt	(cmt, cmtmulti)
~	blanks	blank	none, will be ignored
\t	blanks	blank	none, will be ignored
\n	blanks	blank	none, will be ignored
\r	blanks	blank	none, will be ignored

(b) Lexemes:

```
\t | \t | if | \t | ( | b | \t | == | \t | 1 | ) | \t | { | \n |
\t | \t | \t | x | \t | = | \t | x | \t | + | \t | 1 | ; | \n |
\t | \t | } | \t | else | \t | { | \n |
\t | \t | \t | x | \t | = | \t | x | \t | - | \t | 1 | ; | \n |
\t | \t | } | \n |
```

Symbol sequence (with blanks ignored):

(key, if) (sym, lpar) (id, b) (rel, eq) (num, 1) (sym, rpar) (sym, lbrace) (id, x) (sym, assignment)
(id, x) (aop, plus) (num, 1) (sym, semicolon) (sym, rbrace) (key, else) (sym, lbrace) (id, x) (sym,
assignment) (id, x) (aop, minus) (num, 1) (sym, semicolon) (sym, rbrace)

Exercise 2

(3 Points)

- Provide regular expressions for comments (`//` and `/* ... */`) in *WHILE*. Please keep in mind that `*` as well as `/` may also occur inside of comments. Thus `/*foo * bar / */` is a valid comment. Note that single-line comments are terminated by a newline symbol `\n` (Linux), carriage return `\r` (Mac OS) or `\r\n` (Windows).
- Provide a regular expression capturing numbers in scientific notation, e.g. `-17.42e+23`. To be more precise, a number in scientific notation consists of a floating point number with an optional sign followed by `e` followed by an integer number which may be preceded by an optional sign. In case the floating point number is an integer, the dot may be omitted. Furthermore, if it is less than one, an initial zero may be omitted. Thus, `.3e-8` and `+42e0` are valid numbers in scientific notation.
- Derive *one* NFA that accepts $w \in L_{\text{token}}$, where token is either a comment or a number in scientific notation as defined in the previous tasks.
- Solve the simple matching problem on the NFA from above for the input string `-17.42e+23`.
- For a regular expression r and natural numbers n, m with $n \leq m$ we define a *repetition operator* $r^{[n,m]}$ to denote n to m occurrences of r . For instance, $L(a^{[1,3]}) = \{a, aa, aaa\}$. Prove or disprove that for each regular expression with repetition operators, there exists a regular expression without repetition operators.

Solution: _____

Let Ω be the alphabet of all relevant characters, $A := \Omega \setminus \{*\}$ and $B := \Omega \setminus \{*, /\}$. Moreover, we write $\{a_1, \dots, a_n\}$ as a shortcut for $a_1|a_2|\dots|a_n$.

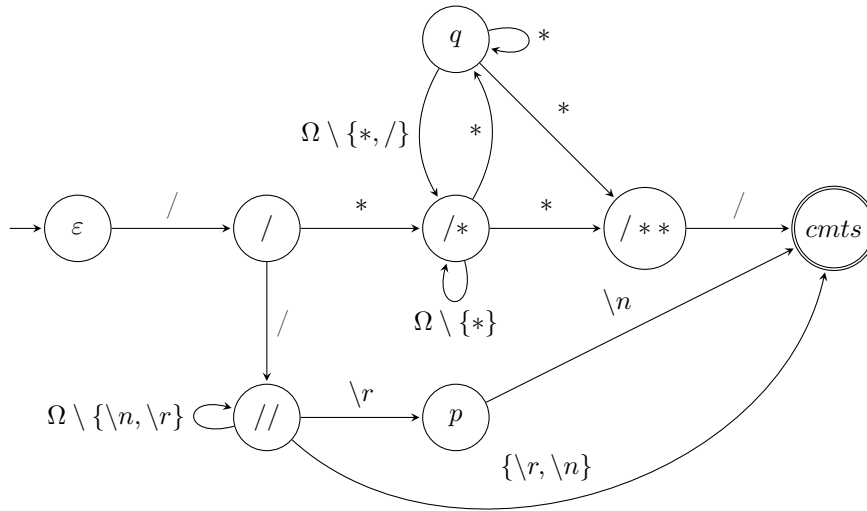
(a)

$$\begin{aligned}\mathcal{L}_{\text{single_cmt}} &:= //(\Omega \setminus \{r, \backslash n\})^*(\backslash n|\backslash r|\backslash r\backslash n) \\ \mathcal{L}_{\text{multi_cmt}} &:= /*A^*(BA^*+)^* */ \\ \mathcal{L}_{\text{comment}} &:= \mathcal{L}_{\text{single_cmt}} | \mathcal{L}_{\text{multi_cmt}}\end{aligned}$$

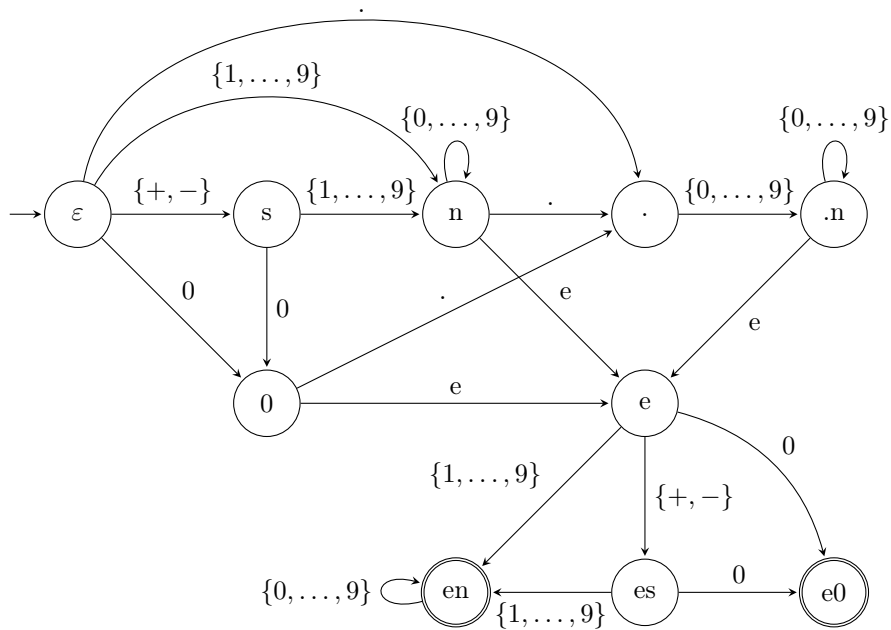
(b) Let $N := \{1, \dots, 9\}\{0, \dots, 9\}^*$ and $S := \{+, -, \epsilon\}$. Then

$$\mathcal{L}_{\text{scientific}} := ((\epsilon|S|0|S|N) \cdot \{0, \dots, 9\}\{0, \dots, 9\}^*|S|N|S|0) \epsilon S (0 + N)$$

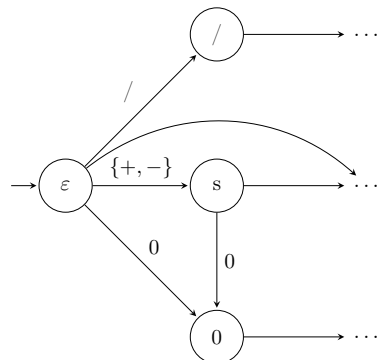
- (c) We use sets as transition labels whenever there is a corresponding transition for every element of that set. The NFA for a comment looks as follows:



The NFA for a number in scientific notation looks as follows:



Combining both NFAs yields the NFA we are looking for:



- (d) The NFA provided in (b) is comfortably deterministic. Still, the NFA method performs the powerset construction on the fly:

$(\{\varepsilon\}, -17.42e+23) \vdash (\{s\}, 17.42e+23) \vdash (\{n\}, 7.42e+23) \vdash (\{n\}, .42e+23) \vdash (\{.\}, 42e+23) \vdash (\{.n\}, 2e+23) \vdash (\{.n\}, e+23) \vdash (\{e\}, +23) \vdash (\{es\}, 23) \vdash (\{en\}, 3) \vdash (\{en\}, \varepsilon)$

- (e) We prove that for each regular expression with a repetition operator there exists an equivalent regular expression without repetition operator.

We have to distinguish two cases:

- $n = m$: here the regular expression $r^{[n,n]}$ can be written as r^n . Correctness is shown by complete induction on n . For $n = 1$, $r^{[1,1]} = r^1 = r$ is immediate. Assume, as induction hypothesis, that for an arbitrary, but fixed $n \in \mathbb{N}$, we have $r^{[n,n]} = r^n$. Now, consider the case $r^{[n+1,n+1]}$. By definition, this corresponds to $n + 1$ occurrences of r , i.e. $r^{[n+1,n+1]} = r^{[n,n]}r$. Thus

$$r^{[n+1,n+1]} = r^{[n,n]}r \stackrel{I.H.}{=} r^n r = r^{n+1}$$

- $n < m$: here the regular expression $r^{[n,m]}$ can be written as $r^n \mid r^{n+1} \mid \dots \mid r^{m-1} \mid r^m$. This construction is correct, because $r^{[n,m]} = r^{[n,n]} \mid r^{[n+1,n+1]} \mid \dots \mid r^{[m,m]}$ and, by the first case, $r^{[k,k]} = r^k$ for each natural number k .

Exercise 3

(5 Points)

In this exercise we make the first steps towards building a lexer. The task of a lexer is to read an input string and return a sequence of symbols. For now, we start by building deterministic finite automata that recognise particular tokens.

On the course webpage (<https://moves.rwth-aachen.de/teaching/ss-16/cc/>), we provide you with a framework which contains the essential class definitions and method declarations for this task. The easiest way to work with that framework is to simply import the files as an existing project into Eclipse. Parts which need implementation are marked with `TODO`.

- Implement the class `AbstractDFA.java` representing an arbitrary DFA with its states and transitions. You may use `Pair.java` in the `helper` module.
- Implement the class `DFA.java` which is instantiated with a string. This class should represent a DFA that recognises exactly the given string.
- Implement the class `CommentDFA.java` that recognises single-line and multi-line comments.

For testing, you find several small examples in the `test` directory. You can compile the program with

```
$javac -d bin -sourcepath src src/Main.java
```

and then run the examples with

```
$java -cp bin Main tests/test1.txt
```

The results on the examples should be:

```
$java -cp bin Main tests/test1.txt
input: while
WHILE: true
COMMENT: false
```

```
$java -cp bin Main tests/test2.txt
input: While
WHILE: false
COMMENT: false
```

```
$java -cp bin Main tests/test3.txt
input: /**while*/
WHILE: false
COMMENT: true
```

```
$java -cp bin Main tests/test4.txt
input: /* */
WHILE: false
COMMENT: false
```

```
$java -cp bin Main tests/test5.txt
input: //foo
WHILE: false
COMMENT: true
```

Solution: _____

For the solution see the code in `While2JasminCompilerSolution`.