

---

**TD1 - Friday, September 25**


---

## 1 Lexical Analysis

### 1.1 Regular expressions and finite automata

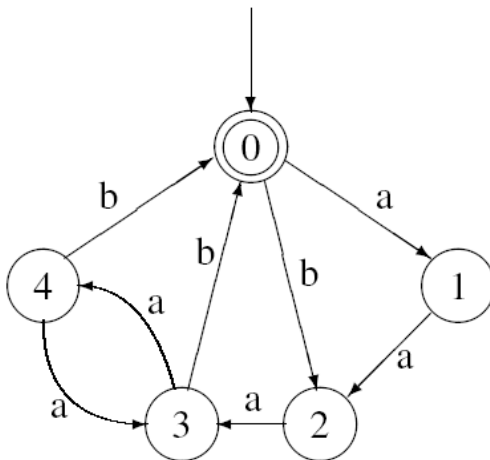
**Exercise 1** Describe using regular expressions, the following :

- the keyword `if`
- a variable name in C
- an integer number
- a floating point number ; examples : `2.76`, `-5.`, `.42`, `5e + 4`, `11.22e - 3`.
- an integer number different from `42`.
- an integer number strictly greater than `42`.

**Exercise 2** Given the regular expression  $((a|b)(a|bb))^*$  :

- Construct an equivalent NFA for it.
- Convert this NFA to a DFA.

**Exercise 3** Minimize the following DFA :



**Exercise 4** Construct a DFA that recognises balanced sequences of parenthesis with a maximal nesting depth of 3, e.g.,  $\epsilon$ ,  $()()$ ,  $((()))$  or  $((()))()()$  but not  $((()))()$  or  $((())((())))$ .

**Exercise 5** Given that binary number strings are read with the most significant bit first and may have leading zeroes, construct DFAs for each of the following languages :

- Binary number strings that represent numbers that are multiples of 4, e.g., `0`, `100` and `10100`.
- Binary number strings that represent numbers that are multiples of 5, e.g., `0`, `101`, `10100` and `11001`.
- Given a number  $n$ , what is the minimal number of states needed in a DFA that recognises binary numbers that are multiples of  $n$ ?

## 1.2 Lexers and lexer generators

If you have solved the exercises in the previous section, then you know how we can convert a language description written as a regular expression into an efficiently executable representation (a DFA). Now, we want to do something more : a program for lexical analysis, i.e. a lexer. It has to distinguish between several different types of tokens. For this exercise consider that the tokens can be : INTEGER, FLOAT, keyword IF, VARIABLE. Each of these are described by its own regular expression as in exercise 1. If there are several ways to split the input into legal tokens, the lexer has to decide which of these it should use. The simplest (dumb) approach would be to generate a DFA for each token definition and apply the DFAs one at a time to the input. Think about a smarter and faster way to generate a single DFA and test for all the tokens simultaneously. For this, use a principle similar to the previous section :

- Create NFAs for each regular expression involved.
- What are the accepting states of these NFAs ?
- Think about a way to combine them in a single NFA.
- Convert the combined NFA to a DFA
- Can the same accepting state in the DFA accept several different token types? If so, give a priority based scheme for solving this problem.
- When we described minimisation of DFAs, we used two initial groups, one for accepting states and one for non-accepting states. How many initial groups do we have to use now?