Lab 1 - Tiger Language

Setup

```
$ mkdir Labs1
$ cd Labs1
$ wget -qO- www.sifflez.org/lectures/compil/lab1/dragon-tiger.tar.gz | tar zxv
 x dragon-tiger/
 x dragon-tiger/Makefile
 x dragon-tiger/src/
 x dragon-tiger/src/driver/
 x dragon-tiger/src/driver/.gitignore
 x dragon-tiger/src/driver/dtiger
 x dragon-tiger/src/runtime/
 x dragon-tiger/src/runtime/posix/
 x dragon-tiger/src/runtime/posix/libruntime.a
 x dragon-tiger/src/runtime/posix/.gitignore
$ cd dragon-tiger
```

Write a Tiger program, hello.tig, that prints the string "Hello World!" followed by a new line character to the standard output.

```
let
    function print_string(s: string) =
        print(s)
    in
        print_string("Hello World!\n")
end
```

Complete the following program and save it in a fibonacci.tig file.

```
let
    function fibonacci(n: int): int =
        if n = 0 then
            1
        else if n = 1 then
            1
        else
            fibonacci(n-1) + fibonacci(n-2)

    in
        fibonacci(10) (* You can replace 10 with any value for nth term you
want to compute *)
end
```

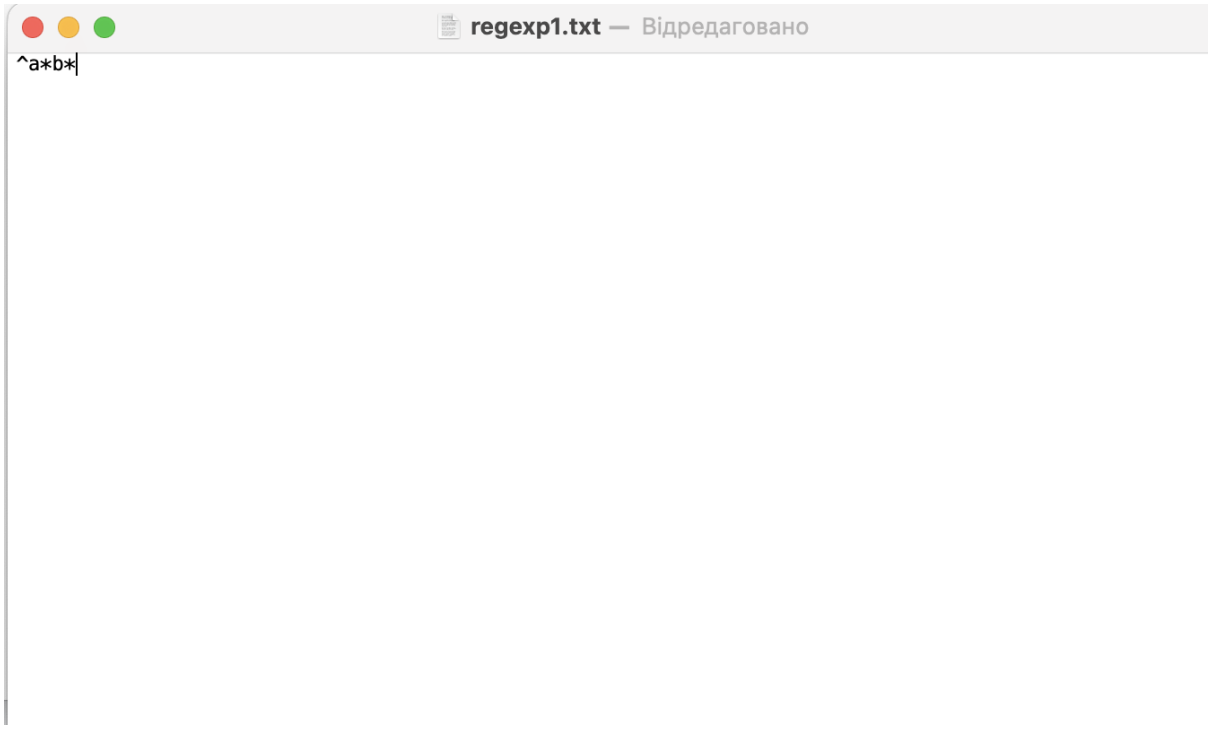
Here's the completed program in read_unsigned.tig:

```
let
    /* Read a positive integer from the standard input.
       Returns -1 on error */
```

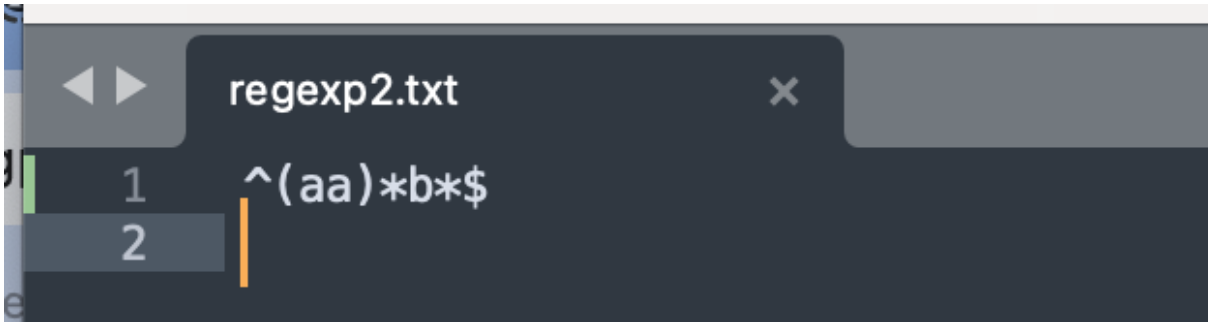
```
    function read_unsigned() : int =
        let
            function is_digit(c: string): bool =
                ord("0") <= ord(c) andalso ord(c) <= ord("9")

            function read_line_acc(acc: int): int =
                let
                    val c = getchar()
                in
                    if c = "" then  (* EOF reached *)
                        if acc = 0 then
                            -1
                        else
                            acc
                    else if c = "\n" then
                        if acc = 0 then
                            -1
                        else
                            acc
                    else if is_digit(c) then
                        read_line_acc(acc * 10 + (ord(c) - ord("0")))
                    else
                        -1
                end

        in
            read_line_acc(0)
        end

    var a : int := read_unsigned()
in
    print_int(a*2);
    print("\n")
end
```

Regular Expressions and Finite Automata

File 1:



File 2:



Result:

```
$ echo "aaaaab" | grep -E -x --color "$(cat regexp1.txt)"
aaaaab
```

Automata Determinisation

What is the language accepted by the automaton in the figure below?

This automaton accepts strings over the alphabet {x, y, z, ε} that begin in state 1 and end in an accepting state (which needs to be specified but isn't shown here; assuming state 4 and state 7 are accepting):

- Starting from state 1, ε-transitions allow skipping directly to states 2, 3, or 4 without consuming any input symbol.
- Similarly, there are ε-transitions between states 5, 6, and 7.
- Additionally, the transitions labeled with x, y, and z form paths from state 1 to 5, 2 to 6, and 1 to 6, respectively.

The language accepted by this automaton includes strings such as:

- x, which leads from state 1 to 5, and potentially continuing with ε to states 6 and 7.
- yz, starting from state 1 (through ε to 2), taking y to state 6, then z could potentially loop back to 6 or move to 7.
- Any combination that can be formed using these transitions while ending in state 4 or 7.

Show that it is not deterministic.

This automaton is non-deterministic because:

1. There are ε-transitions (transitions that do not consume any input symbol) allowing multiple transitions from a single state without consuming input.
2. Multiple transitions for the same input from a given state (e.g., from state 1 to 5 on input x and from 1 to 2 on ε, and then potentially to others like 6 on z).

Determinise it.

To determinize this automaton, we need to eliminate all ε-transitions and handle multiple transitions for the same input. We use the subset construction method, where each state in the deterministic automaton (DFA) represents a set of states from the NFA:

1. Start state: {1, 2, 3, 4} due to ε-closure from state 1.
2. On input x:
    - From state 1: leads to state 5.
    - ε-closure of state 5 includes states 5, 6, 7.
3. On input y:

- From state 2: leads to state 6.
- ε-closure of state 6 includes states 6, 7.

4. On input z:
    - From state 1: leads to state 6.
    - ε-closure of state 6 includes states 6, 7.

DFA States:

- S0: {1, 2, 3, 4}
- S1: {5, 6, 7} (from S0 on x or z)
- S2: {6, 7} (from S0 on y and also from S1 on y or z)

DFA Transitions:

- S0 --x--> S1
- S0 --z--> S1
- S0 --y--> S2
- S1 --y--> S2
- S1 --z--> S2