Principi programskih jezikov

First exam, 10. june 2024

1	
2	
3	
4	
Σ	

Name and surname

Student ID

Instructions

- **Do not open the exam** before the test starts.
- **Before you begin:**
  - Write your name and student ID on the exam in capitals.
  - Place a photo ID and your student card on the desk.
  - Turn off your phone and put it in your bag.
  - Log in to the online classroom where you will submit your answers.
- Allowed accessories: pen, eraser, files on Moodle, and any literature.
- Enter all solutions in the online quiz.
- If you need anything, ask the assistant, not your neighbors.
- **Do not leave your place during the exam** without permission.
- Your exam will be taken away **without further warnings** if you:
  - communicate with anyone except the assistant,
  - give a piece of paper or some other object to anyone,
  - move your materials so that someone else can see it,
  - cheat in any other way, or help anyone else cheat,
  - have a phone or some other electronic device in a visible place.
- **When exam ends:**
  - When the assistant declares the test has finished, **immediately** stop and close the exam.
  - **Do not get up**, but wait for the assistant to collect **all** exams.
  - **You must submit your exam.**
- Duration of exam is 120 minutes. The time is written on the whiteboard.
- Estimated grading criteria:

  ≥ 90 points, grade 10

  ≥ 80 points, grade 9

  ≥ 70 points, grade 8

  ≥ 60 points, grade 7

  ≥ 50 points, grade 6

Question 1 (21 points)

a) (7 points) In a programming language with subtypes we have $\texttt{bool} \leq \texttt{int}$. We are given record types

$$\rho = \{a : \texttt{bool}\} \to \{b : \texttt{bool}; c : \texttt{int}\}$$
$$\sigma = \{a : \texttt{int}\} \to \{b : \texttt{bool}; c : \texttt{int}\}$$
$$\tau = \{a : \texttt{bool}\} \to \{b : \texttt{int}\}$$
$$\psi = \{\} \to \{b : \texttt{int}; c : \texttt{int}\}$$

We are using width and depth subtyping for record types. In the table below, for each pair, indicate with YES/NO if the type in the row is a subtype of the type in the column:

$\leq$	ρ	σ	τ	ψ
ρ				
σ				
τ				
ψ				

b) (7 points) The OCaml standard library contains **right**-associative binary operators `@@` and `::` with *principal types*:

$$\texttt{(@@)} : (\alpha \to \beta) \to \alpha \to \beta$$
$$\texttt{(::)} : \alpha \to \alpha\ \texttt{list} \to \alpha\ \texttt{list}$$

The operator `@@` has lower precedence than the `::` operator. Infer (by hand) the principal type of the expression below

```
fun (a, b) -> a @@ a @@ 1 :: b
```

c) (7 points) In the λ-calculus, implement the function `scott`, which converts a Church numerals (`0'`, `1'`, `2'`,...)

```
0' := ^ f x . x ;
1' := ^ f x . f x ;
2' := ^ f x . f (f x) ;
3' := ^ f x . f (f (f x)) ;
4' := ^ f x . f (f (f (f x))) ;
5' := ^ f x . f (f (f (f (f x)))) ;
succ' := ^ n . ^ f x . f (n f x) ;
```

into Scott-Church numeral (`0`, `1`, `2`,...)

```
0 := ^ f x . x ;
1 := ^ f x . f 0 x ;
2 := ^ f x . f 1 (f 0 x) ;
3 := ^ f x . f 2 (f 1 (f 0 x)) ;
4 := ^ f x . f 3 (f 2 (f 1 (f 0 x))) ;
5 := ^ f x . f 4 (f 3 (f 2 (f 1 (f 0 x)))) ;
succ := ^ n . ^ f x . f n (n f x) ;
```

Question 2 (21 points)

In OCaml, we want to implement a simulator for the Elbonian stack machine. The simulator will execute commands (of type `instruction`), which are located in ROM (`rom : int -> instruction`), and read numbers from the input (`input : int list`) with the command `READ`. When program execution is terminated with the `EXIT` command, the program returns the top of the stack. A stack is represented by a list (`stack : int list`), where the top of the stack is the first element of the list. The state of the machine is represented by the `machine_state` record type, which holds the unread input, ROM, stack, and the instruction pointer IP (`instruction_pointer : int`).
Instruction set with descriptions:

```
type instruction =
  | NOOP (* no operation, IP := IP + 1 *)
  | PUSH of int (* push integer constant onto stack, IP := IP + 1 *)
  | ADD (* push sum of top two elements of stack, IP := IP + 1 *)
  | SUB (* push difference of top two elements of stack, IP := IP + 1 *)
  | EQ (* push 1 if top two elements of stack are equal otherwise push 0, IP := IP + 1 *)
  | LT (* push 1 if top top element of stack is less than the second element
    otherwise push 0, IP := IP + 1 *)
  | JMP of int (* relative jump IP := IP + rel *)
  | JMPZ of int (* jump if zero on the stack IP := IP + rel, otherwise IP := IP + 1 *)
  | EXIT (* stop execution, returning the top element of the stack *)
  | READ (* push read integer, IP := IP + 1 *)
  | DUP (* duplicate top element on the stack, IP := IP + 1 *)

type machine_state = {
  input : int list;
  rom : int -> instruction;
  stack : int list;
  instruction_pointer : int;
}
```

Complete the implementation of the function `run : machine_state -> int`. Use the helper functions provided below (see file `asm_unfinished.ml` on Moodle):

```
let pop state =
  match state.stack with
  | x :: stack -> (x, { state with stack })
  | [] -> failwith "empty stack"

let read state =
  match state.input with
  | x :: input -> (x, { state with input })
  | [] -> failwith "empty input"

let double_pop state =
  let x1, state = pop state in
  let x2, state = pop state in
  (x1, x2, state)

let push x state = { state with stack = x :: state.stack }

let increment_instruction_pointer i state =
  { state with instruction_pointer = state.instruction_pointer + i }

let string_of_instruction = function
  | NOOP -> "NOOP" | ADD -> "ADD" | SUB -> "SUB" | EQ -> "EQ"
  | LT -> "LT" | EXIT -> "EXIT" | READ -> "READ" | DUP -> "DUP"
  | PUSH i -> "PUSH " ^ string_of_int i
  | JMP i -> "JMP " ^ string_of_int i
  | JMPZ i -> "JMPZ " ^ string_of_int i

let print_debug_info { input; rom; stack; instruction_pointer } =
  Printf.printf "INFO: IP = %i (%s), stack = [%s], input = [%s]\n"
  instruction_pointer
  (string_of_instruction @@ rom instruction_pointer)
  (String.concat ", " @@ List.map string_of_int stack)
  (String.concat ", " @@ List.map string_of_int input)
```

```
let rec run (state : machine_state) =
  print_debug_info state;
  let incr = increment_instruction_pointer 1 in
  match state.rom state.instruction_pointer with
  | NOOP -> run state
  | _ -> failwith "not implemented"

let romA i = [| PUSH 0; READ; DUP; JMPZ 3; ADD; JMP (-4); JMPZ 1; EXIT |].(i)

let computerA input =
  run { rom = romA; stack = []; input; instruction_pointer = 0 }
```

Examples:

```
# run { rom = (fun i -> [| PUSH 42; EXIT |].(i)); stack = []; input = [1; 2];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (PUSH 42), stack = [], input = [1; 2]
INFO: IP = 1 (EXIT), stack = [42], input = [1; 2]
- : int = 42

# run { rom = (fun i -> [| READ; EXIT |].(i)); stack = []; input = [1; 2];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (READ), stack = [], input = [1; 2]
INFO: IP = 1 (EXIT), stack = [1], input = [2]
- : int = 1

# run { rom = (fun i -> [| ADD; EXIT |].(i)); stack = [-1; 1; 2]; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (ADD), stack = [-1; 1; 2], input = []
INFO: IP = 1 (EXIT), stack = [0; 2], input = []
- : int = 0

# run { rom = (fun i -> [| SUB; EXIT |].(i)); stack = [-1; 1; 2]; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (SUB), stack = [-1; 1; 2], input = []
INFO: IP = 1 (EXIT), stack = [-2; 2], input = []
- : int = -2

# run { rom = (fun i -> [| EQ; EXIT |].(i)); stack = [-1; 1; 2]; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (EQ), stack = [-1; 1; 2], input = []
INFO: IP = 1 (EXIT), stack = [0; 2], input = []
- : int = 0

# run { rom = (fun i -> [| EQ; EXIT |].(i)); stack = [1; 1; 2]; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (EQ), stack = [1; 1; 2], input = []
INFO: IP = 1 (EXIT), stack = [1; 2], input = []
- : int = 1

# run { rom = (fun i -> [| EQ; EXIT |].(i)); stack = [1; 1; 2]; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (EQ), stack = [1; 1; 2], input = []
INFO: IP = 1 (EXIT), stack = [1; 2], input = []
- : int = 1

# run { rom = (fun i -> [| LT; EXIT |].(i)); stack = [-1; 1; 2]; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (LT), stack = [-1; 1; 2], input = []
INFO: IP = 1 (EXIT), stack = [1; 2], input = []
- : int = 1

# run { rom = (fun i -> [| JMP 2; READ; EXIT |].(i)); stack = []; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (JMP 2), stack = [], input = []
INFO: IP = 2 (EXIT), stack = [], input = []
Exception: Failure "empty stack".

# run { rom = (fun i -> [| JMP 2; READ; EXIT |].(i)); stack = [100]; input = [];
 instruction_pointer = 0 } ;;
```

```
INFO: IP = 0 (JMP 2), stack = [100], input = []
INFO: IP = 2 (EXIT), stack = [100], input = []
- : int = 100

# run { rom = (fun i -> [| JMPZ 2; READ; EXIT |].(i)); stack = [0; 100]; input = [-1];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (JMPZ 2), stack = [0; 100], input = [-1]
INFO: IP = 2 (EXIT), stack = [100], input = [-1]
- : int = 100

# run { rom = (fun i -> [| JMPZ 2; READ; EXIT |].(i)); stack = [3; 100]; input = [-1];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (JMPZ 2), stack = [3; 100], input = [-1]
INFO: IP = 1 (READ), stack = [100], input = [-1]
INFO: IP = 2 (EXIT), stack = [-1; 100], input = []
- : int = -1

# run { rom = (fun i -> [| DUP; EXIT |].(i)); stack = [3; 100]; input = [];
 instruction_pointer = 0 } ;;
INFO: IP = 0 (DUP), stack = [3; 100], input = []
INFO: IP = 1 (EXIT), stack = [3; 3; 100], input = []
- : int = 3

# computerA [42; -2; 60; 0; 1337];;
INFO: IP = 0 (PUSH 0), stack = [], input = [42; -2; 60; 0; 1337]
INFO: IP = 1 (READ), stack = [0], input = [42; -2; 60; 0; 1337]
INFO: IP = 2 (DUP), stack = [42; 0], input = [-2; 60; 0; 1337]
INFO: IP = 3 (JMPZ 3), stack = [42; 42; 0], input = [-2; 60; 0; 1337]
INFO: IP = 4 (ADD), stack = [42; 0], input = [-2; 60; 0; 1337]
INFO: IP = 5 (JMP -4), stack = [42], input = [-2; 60; 0; 1337]
INFO: IP = 1 (READ), stack = [42], input = [-2; 60; 0; 1337]
INFO: IP = 2 (DUP), stack = [-2; 42], input = [60; 0; 1337]
INFO: IP = 3 (JMPZ 3), stack = [-2; -2; 42], input = [60; 0; 1337]
INFO: IP = 4 (ADD), stack = [-2; 42], input = [60; 0; 1337]
INFO: IP = 5 (JMP -4), stack = [40], input = [60; 0; 1337]
INFO: IP = 1 (READ), stack = [40], input = [60; 0; 1337]
INFO: IP = 2 (DUP), stack = [60; 40], input = [0; 1337]
INFO: IP = 3 (JMPZ 3), stack = [60; 60; 40], input = [0; 1337]
INFO: IP = 4 (ADD), stack = [60; 40], input = [0; 1337]
INFO: IP = 5 (JMP -4), stack = [100], input = [0; 1337]
INFO: IP = 1 (READ), stack = [100], input = [0; 1337]
INFO: IP = 2 (DUP), stack = [0; 100], input = [1337]
INFO: IP = 3 (JMPZ 3), stack = [0; 0; 100], input = [1337]
INFO: IP = 6 (JMPZ 1), stack = [0; 100], input = [1337]
INFO: IP = 7 (EXIT), stack = [100], input = [1337]
- : int = 100
```

(Empty page)

Question 3 (28 points)

a) (21 points) Extended Fibonacci sequence $\ldots, F(-3), F(-2), F(-1), F(0), F(1), \ldots$ is defined as

$$F(0) = 0, \qquad F(1) = 1, \qquad F(i+2) = F(i+1) + F(i).$$

i	$\ldots$	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	$\ldots$
$F(i)$	$\ldots$	13	-8	5	-3	2	-1	1	0	1	1	2	3	5	8	13	$\ldots$

Prove *partial* correctness of the following program, where N is an integer.

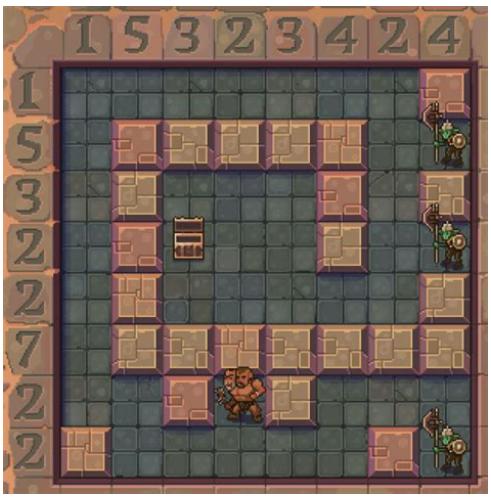
```
{ true }
x := 1 ;
y := 0 ;
z := 0 ;
if N > 0 then
  while not (y = N) do
    z := z + 1 ;
    w := x ;
    x := y ;
    y := y + w ;
  done
else
  while not (y = N) do
    z := z - 1 ;
    w := y ;
    y := x ;
    x := w - x ;
  done
end
{ N = F(z) }
```

b) (7 points) Prove or disprove the *total* correctness of the above program.

(Empty page)

Question 4 (35 points)

An Elbonian civil engineer was given "blueprints" for making dungeons. The dungeon is a rectangular basement with "thick" walls (see the pictures). The engineer is confused, because the architect only gave him the number of necessary walls in each "row/column", and where the beasts/treasures will be. Due to the construction method, the two walls cannot touch only at the corners. In the two pictures below, we can see the possible layout of the walls for two different "blueprints". Make his job easier with the help of Prolog.



a) (14 points) With the help of the predicate `nice_corners/4`, which is valid when

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \neq \begin{bmatrix} A & B \\ C & D \end{bmatrix} \neq \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

```
nice_corners(A, B, C, D) :-
  (A = B; C = D, A = C; B = D), !.
```

define a predicate `nice_corners(M)`, which is valid when a matrix (list of lists) `M` does not contain a sub-matrices $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ in $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Examples:

```
?- nice_corners([[0,1], [0,1]]).
true ;
false.
?- nice_corners([[0,1], [1,0]]).
false.
?- nice_corners([[1,0], [0,1]]).
false.
?- nice_corners([[1,1,0], [0,1,0], [0,1,1]]).
true ;
false.
?- nice_corners([[1,1,0], [0,1,0], [0,1,1]]).
true ;
false.
?- nice_corners([[1,0,1], [0,1,0], [0,1,1]]).
false.
?- nice_corners([[1,0,1], [0,0,1], [0,1,0]]).
false.
```

Helper predicates are allowed.

b) (21 points) Define a predicate `dungeon(M, H, V)`, which is valid when:

- the matrix (list of lists) `M` is a binary,
- list `H` is a sum of all rows of `M`,
- list `V` is a sum of all columns of `M`,
- all rows are of the same length,
- all columns are of the same length.

Helper predicates are allowed.
Examples.

```
?- dungeon(M, [1,2,3], [3, 2, 1]).
M = [[1, 0, 0], [1, 1, 0], [1, 1, 1]].

?- dungeon(M, [1,2,4,3], [4,3,2,1]).
M = [[1, 0, 0, 0], [1, 1, 0, 0], [1, 1, 1, 1], [1, 1, 1, 0]].

?- dungeon(M, [1,1,3,3], [3,2,2,1]), nice_corners(M), maplist(portray_clause, M).
[0, 0, 0, 1].
[1, 0, 0, 0].
[1, 1, 1, 0].
[1, 1, 1, 0].
M = [[0, 0, 0, 1], [1, 0, 0, 0], [1, 1, 1, 0], [1, 1, 1, 0]] ;
[1, 0, 0, 0].
[1, 0, 0, 0].
[1, 1, 1, 0].
[0, 1, 1, 1].
M = [[1, 0, 0, 0], [1, 0, 0, 0], [1, 1, 1, 0], [0, 1, 1, 1]] ;
false.

?- M =
[[0, _, 0, _, _, _, _, _],
 [_, _, _, _, _, _, _, _],
 [_, _, _, _, _, 0, _, _],
 [_, _, _, _, _, _, _, _],
 [_, _, _, _, _, _, _, _],
 [_, _, _, _, _, _, _, 0],
 [_, _, _, _, _, _, _, _],
 [_, _, _, _, _, _, _, 0]],
 dungeon(M, [6,3,1,3,6,1,6,0], [0,6,1,5,3,2,3,6]), maplist(portray_clause, M).
[0, 1, 0, 1, 1, 1, 1, 1].
[0, 1, 0, 1, 0, 0, 0, 1].
[0, 0, 0, 0, 0, 0, 0, 1].
[0, 1, 0, 1, 0, 0, 0, 1].
[0, 1, 0, 1, 1, 1, 1, 1].
[0, 1, 0, 0, 0, 0, 0, 0].
[0, 1, 1, 1, 1, 0, 1, 1].
[0, 0, 0, 0, 0, 0, 0, 0].
M = [[0, 1, 0, 1, 1, 1, 1, 1]|...] ;
[0, 1, 0, 1, 1, 1, 1, 1].
[0, 1, 0, 1, 0, 0, 0, 1].
[0, 0, 0, 0, 0, 0, 0, 1].
[0, 1, 0, 1, 0, 0, 0, 1].
[0, 1, 1, 1, 1, 0, 1, 1].
[0, 1, 0, 0, 0, 0, 0, 0].
[0, 1, 0, 1, 1, 1, 1, 1].
[0, 0, 0, 0, 0, 0, 0, 0].
M = [[0, 1, 0, 1, 1, 1, 1, 1]|...].
```