

Homework 1

CSE307, Spring 2026

Minseok Jeon

Problem 1 Write a function

```
prime : int → bool
```

that checks whether a number is prime (n is prime if and only if n is its own smallest divisor except for 1). For example,

```
prime 2 = true
prime 3 = true
prime 4 = false
prime 17 = true
```

Problem 2 Write a function

```
range : int → int → int list
```

that takes two integers n and m , and creates a list of integers from n to m . For example, `range 3 7` produces `[3;4;5;6;7]`. When $n > m$, an empty list is returned. For example, `range 5 4` produces `[]`.

Problem 3 Write a function

```
suml : int list list → int
```

which takes a list of lists of integers and sums the integers included in all the lists. For example, `suml [[1;2;3]; []; [-1; 5; 2]; [7]]` produces 19.

Problem 4 Write a function `drop`:

```
drop : 'a list → int → 'a list
```

that takes a list l and an integer n to take all but the first n elements of l . For example,

```
drop [1;2;3;4;5] 2 = [3; 4; 5]
drop [1;2] 3 = []
drop ["C"; "Java"; "OCaml"] 2 = ["OCaml"]
```

Problem 5 Write two functions

```
max : int list → int      min : int list → int
```

that find maximum and minimum elements of a given list, respectively. For example `max [1;3;5;2]` should evaluate to 5 and `min [1;3;2]` should be 1.

Problem 6 Write a higher-order function

```
sigma : (int → int) → int → int → int
```

such that `sigma f a b` computes

$$\sum_{i=a}^b f(i).$$

For instance,

```
sigma (fun x → x) 1 10
```

evaluates to 55 and

```
sigma (fun x → x*x) 1 7
```

evaluates to 140.

Problem 7 Write a higher-order function

```
forall : ('a → bool) → 'a list → bool
```

which decides if all elements of a list satisfy a predicate. For example,

```
forall (fun x → x mod 2 = 0) [1;2;3]
```

evaluates to `false` while

```
forall (fun x → x > 5) [7;8;9]
```

is `true`.

Problem 8 Write a function

```
double : ('a → 'a) → 'a → 'a
```

that takes a function of one argument as argument and returns a function that applies the original function twice. For example,

```
# let inc x = x + 1;;
val inc : int → int = <fun>
# let mul x = x * 2;;
val mul : int → int = <fun>
# (double inc) 1;;
- : int = 3
# (double inc) 2;;
- : int = 4
# ((double double) inc) 0;;
- : int = 4
# ((double (double double)) inc) 5;;
- : int = 21
# (double mul) 1;;
- : int = 4
# (double double) mul 2;;
- : int = 32
```

Problem 9 Binary trees can be defined as follows:

```
type btree =
  Empty
  | Node of int * btree * btree
```

For example, the following t1 and t2

```
let t1 = Node (1, Empty, Empty)
let t2 = Node (1, Node (2, Empty, Empty), Node (3, Empty, Empty))
```

are binary trees. Write the function

```
mem : int → btree → bool
```

that checks whether a given integer is in the tree or not. For example, `mem 1 t1` evaluates to *true*, and `mem 4 t2` evaluates to *false*.

Problem 10 Consider the inductive definition of binary trees:

$$\frac{_}{\bar{n}} \quad n \in \mathbb{Z} \qquad \frac{t}{(t, \mathbf{nil})} \qquad \frac{t}{(\mathbf{nil}, t)} \qquad \frac{t_1 \quad t_2}{(t_1, t_2)}$$

which can be defined in OCaml as follows:

```
type btree =
  | Leaf of int
  | Left of btree
  | Right of btree
  | LeftRight of btree * btree
```

For example, binary tree $((1, 2), \mathbf{nil})$ is represented by

```
Left (LeftRight (Leaf 1, Leaf 2))
```

Write a function that exchanges the left and right subtrees all the way down. For example, mirroring the tree $((1, 2), \mathbf{nil})$ produces $(\mathbf{nil}, (2, 1))$; that is,

```
mirror (Left (LeftRight (Leaf 1, Leaf 2)))
```

evaluates to

```
Right (LeftRight (Leaf 2, Leaf 1))
```

Problem 11 Natural numbers are defined inductively:

$$\frac{_}{0} \qquad \frac{n}{n + 1}$$

In OCaml, the inductive definition can be defined by the following data type:

```
type nat = ZERO | SUCC of nat
```

For instance, `SUCC ZERO` denotes 1 and `SUCC (SUCC ZERO)` denotes 2. Write two functions that add and multiply natural numbers:

```
natadd : nat → nat → nat      natmul : nat → nat → nat
```

For example,

```

# let two = SUCC (SUCC ZERO);;
val two : nat = SUCC (SUCC ZERO)
# let three = SUCC (SUCC (SUCC ZERO));;
val three : nat = SUCC (SUCC (SUCC ZERO))
# natmul two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))))
# natadd two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))

```

Problem 12 Consider the following propositional formula:

```

type formula =
| True
| False
| Not of formula
| AndAlso of formula * formula
| OrElse of formula * formula
| Imply of formula * formula
| Equal of exp * exp
and exp =
| Num of int
| Plus of exp * exp
| Minus of exp * exp

```

Write the function

```
eval : formula → bool
```

that computes the truth value of a given formula. For example,

```
eval (Imply (Imply (True, False), True))
```

evaluates to *true*, and

```
eval (Equal (Num 1, Plus (Num 1, Num 2)))
```

evaluates to *false*.

Problem 13 Write a function

```
diff : aexp × string → aexp
```

that differentiates the given algebraic expression with respect to the variable given as the second argument. The algebraic expression **aexp** is defined as follows:

```

type aexp =
| Const of int
| Var of string
| Power of string * int
| Times of aexp list
| Sum of aexp list

```

For example, $x^2 + 2x + 1$ is represented by

```
Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]
```

and differentiating it (w.r.t. “x”) gives $2x + 2$, which can be represented by

```
Sum [Times [Const 2; Var "x"]; Const 2]
```

Note that the representation of $2x + 2$ in `aexp` is not unique. For instance, the following also represents $2x + 2$:

```
Sum
```

```
[Times [Const 2; Power ("x", 1)];  
 Sum  
   [Times [Const 0; Var "x"];  
    Times [Const 2; Sum [Times [Const 1]; Times [Var "x"; Const 0]]]];  
 Const 0]
```

Problem 14 Consider the following expressions:

```
type exp = X  
| INT of int  
| ADD of exp * exp  
| SUB of exp * exp  
| MUL of exp * exp  
| DIV of exp * exp  
| SIGMA of exp * exp * exp
```

Implement a calculator for the expressions:

```
calculator : exp → int
```

For instance,

$$\sum_{x=1}^{10} (x * x - 1)$$

is represented by

```
SIGMA(INT 1, INT 10, SUB(MUL(X, X), INT 1))
```

and evaluating it should give 375.

Problem 15 Consider the following language:

```
type exp = V of var  
| P of var * exp  
| C of exp * exp  
and var = string
```

In this language, a program is simply a variable, a procedure, or a procedure call. Write a checker function

```
check : exp → bool
```

that checks if a given program is well-formed. A program is said to be *well-formed* if and only if the program does not contain free variables; i.e., every variable name is bound by some procedure that encompasses the variable. For example, well-formed programs are:

- $P ("a", V "a")$
- $P ("a", P ("a", V "a"))$
- $P ("a", P ("b", C (V "a", V "b")))$
- $P ("a", C (V "a", P ("b", V "a")))$

Ill-formed ones are:

- $P ("a", V "b")$
- $P ("a", C (V "a", P ("b", V "c")))$
- $P ("a", P ("b", C (V "a", V "c")))$