Schedule

SCHEDULE

w	date	topic
~~7~~	~~November 25~~	~~introduction~~
8	December 2	higher-order functions, lists, trees
9	December 9	graphs, combinatorics
10	December 16	program reasoning
11	January 13	λ and interpreter
12	January 20	type system
13	January 27	exam part 2

CONTENTS

1. higher-order functions on lists
2. exceptions
3. algebraic data types
4. homework 2

Higher-order Functions

Higher-Order and Anonymous Functions, Map

- (higher-order) function can take functions and return function
- anonymous function is constructed by fun

DEFINITION

$$\text{List.map } f\ [x_1; \cdots ; x_n] = [f\ x_1; \cdots ; f\ x_n]$$

EXAMPLE

```
# let plus1 x = x + 1;;
# List.map plus1 [1; 2; 3];;
- : int list = [2; 3; 4]
# List.map (fun x -> x + 1) [1; 2; 3];;
- : int list = [2; 3; 4]
```


Fold

DEFINITION

$$\begin{aligned} \text{List.fold\_left } (\circ)\ e\ [x_1; \cdots ; x_n] &= (e \circ x_1) \circ \cdots \circ x_n \\ \text{List.fold\_right } (\circ)\ [x_1; \cdots ; x_n]\ e &= x_1 \circ \cdots \circ (x_n \circ e) \end{aligned}$$

EXAMPLE

```
# List.fold_left (+) 0 [1;2;3];;
- : int = 6
# List.fold_left (fun x y -> x - y) 10 [1;2;3];;
- : int = 4
# List.fold_right (@) [[1;2;3]; [4;5]; [6]] [];;
- : int list = [1; 2; 3; 4; 5; 6]
```

EXERCISES

- `sum : int list -> int`
- `concat : 'a list list -> 'a list`

Filter

$$
\begin{aligned}
\text{List.filter even } & [\,] &= &\ [\,] \\
\text{List.filter even } & (4 :: [\,]) &= &\ 4 :: [\,] \\
\text{List.filter even } & (3 :: 4 :: [\,]) &= &\ 4 :: [\,] \\
\text{List.filter even } & (2 :: 3 :: 4 :: [\,]) &= &\ 2 :: 4 :: [\,] \\
\text{List.filter even } & (1 :: 2 :: 3 :: 4 :: [\,]) &= &\ 2 :: 4 :: [\,]
\end{aligned}
$$

where even $x = x \text{ mod } 2 = 0$

```
let rec filter p = function
  | [] ->
  | x :: xs when p x ->
  | _ :: xs ->
```


Partition

$$
\begin{aligned}
\text{partition even } & [\,] &= &\ ([\,], [\,]) \\
\text{partition even } & (3 :: [\,]) &= &\ ([\,], 3 :: [\,]) \\
\text{partition even } & (2 :: 3 :: [\,]) &= &\ (2 :: [\,], 3 :: [\,]) \\
\text{partition even } & (1 :: 2 :: 3 :: [\,]) &= &\ (2 :: [\,], 1 :: 3 :: [\,])
\end{aligned}
$$

```
let rec partition p = function
  | [] ->
  | x :: xs ->
     let ys, zs = partition p xs in
```


Quicksort

Hoare 1960

$$
\begin{aligned}
\text{qsort } [\,] &= [\,] \\
\text{qsort } (x :: xs) &= \text{qsort } [y \mid y \leftarrow xs, y \leqslant x] \ @ \ [x] \ @ \\
& \quad\ \text{qsort } [y \mid y \leftarrow xs, x < y]
\end{aligned}
$$

- based on divide and conquer
- simpler than definition of sorted list

EXERCISE

implement it!

Divide and Conquer

$$
\begin{gathered}
\text{qsort } [3; 5; 2; 4; 1] \\
= \\
\text{qsort } [2; 1] \quad @ \ [3] \ @ \quad \text{qsort } [5; 4] \\
= \\
(\text{qsort } [1] \ @ \ [2] \ @ \ \text{qsort } [\,]) \ @ \ [3] \ @ \ (\text{qsort } [4] \ @ \ [5] \ @ \ \text{qsort } [\,]) \\
= \\
(([\,] \ @ \ [1] \ @ \ [\,]) \ @ \ [2] \ @ \ [\,]) \ @ \ [3] \ @ \ (([\,] \ @ \ [4] \ @ \ [\,]) \ @ \ [5] \ @ \ [\,]) \\
= \\
([1] \ @ \ [2] \ @ \ [\,]) \ @ \ [3] \ @ \ ([4] \ @ \ [5] \ @ \ [\,]) \\
= \\
[1; 2] \ @ \ [3] \ @ \ [4; 5] \\
= \\
[1; 2; 3; 4; 5]
\end{gathered}
$$

Exceptions

Raising Exceptions

- `exception` declares exception
- `raise` raises exception

EXAMPLE

$$\text{assoc } "x" \; [("x", 1); ("y", 2)] = 1$$
$$\text{assoc } "y" \; [("x", 1); ("y", 2)] = 2$$
$$\text{assoc } "z" \; [("x", 1); ("y", 2)] = \text{raise Not\_found}$$

```
exception Not_found
let rec assoc x = function
  | [] ->
  | (y, z) :: _ when x = y ->
  | _ :: alist ->
```


Catching Exceptions

- exception slips through functions
- `try` ... `with` ... pattern-matches exception to catch

EXAMPLE

$$1 + \text{raise Not\_found} = \text{raise Not\_found}$$
$$\text{try } 1 + \text{raise Not\_found with Not\_found} \to 0 = 0$$
$$\text{try } 1 + 2 \text{ with Not\_found} \to 0 = 3$$

Algebraic Data Types

Algebraic Data Types

- algebraic data type is declared by type
- consists of values and data constructors

EXAMPLE

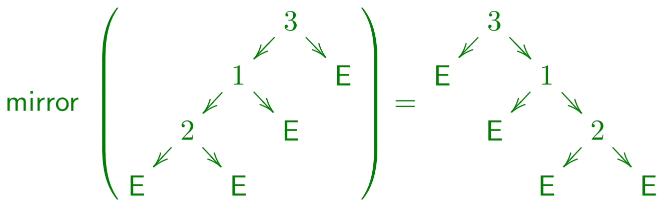
```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

$$\begin{aligned}&\text{Empty}\\&\text{Node}(\text{Empty}, 2, \text{Empty})\\&\text{Node}(\text{Node}(\text{Empty}, 2, \text{Empty}), 1, \text{Empty})\end{aligned}$$

Pattern Matching

EXAMPLE

$$\begin{aligned}&\text{mirror } (\text{Node}(\text{Node}(\text{Node}(\text{Empty}, 2, \text{Empty}), 1, \text{Empty}), 3, \text{Empty}))\\=\ &\text{Node}(\text{Empty}, 3, \text{Node}(\text{Empty}, 1, \text{Node}(\text{Empty}, 2, \text{Empty})))\end{aligned}$$




```
let rec mirror = function
  | Empty ->
  | Node (l, x, r) ->
```


EXERCISES

- `preorder : 'a tree -> 'a list`
- `postorder : 'a tree -> 'a list`

Expressions and Evaluation

EXAMPLE

```
type expr =
  | Const of int
  | Add of expr * expr
  | Mul of expr * expr
```

$$\begin{aligned}&\text{eval}(\text{Const } 2) &&= 2\\&\text{eval}(\text{Mul}(\text{Const } 2, \text{Const } 3)) &&= 6\\&\text{eval}(\text{Add}(\text{Const } 1, \text{Mul}(\text{Const } 2, \text{Const } 3))) &&= 7\end{aligned}$$

```
let rec eval = function
  | Const n ->
  | Add (e1, e2) ->
  | Mul (e1, e2) ->
```

Printing Data Structures

- open in OCaml ≈ import in Java
- module Format includes fprintf, std_formatter, ...

```
# open Format;;

# let rec fprint_expr formatter = function
  | Const n -> fprintf formatter "%d" n
  | Add (e1, e2) ->
      fprintf formatter "(%a + %a)"
        fprint_expr e1 fprint_expr e2
  | Mul (e1, e2) ->
      fprintf formatter "(%a * %a)"
        fprint_expr e1 fprint_expr e2;;

# fprint_expr std_formatter
    (Add (Const 1, Mul (Const 2, Const 3)));;
(1 + (2 * 3))- : unit
```


Installing Printer

```
# #install_printer fprint_expr;;
# Mul (Const 2, Add (Const 3, Const 4));;
- : expr = (2 * (3 + 4))

# #trace eval;;
eval is now traced.
# eval (Mul (Const 2, Add (Const 3, Const 4)));;
eval <-- (2 * (3 + 4))
eval <-- (3 + 4)
eval <-- 4
eval --> 4
eval <-- 3
eval --> 3
eval --> 7
eval <-- 2
eval --> 2
eval --> 14
- : int = 14
```


Homework: Stack-based Virtual Machines

Stack-based VM

- instruction set

```
type instr = Push | Addint | Mulint | Value of int
```

- bytecode compiler

```
# compile (Mul (Const 10, Add (Const 20, Const 30)));;
- : instr list = [Push; Value 10; Push; Value 20;
                  Push; Value 30; Addint; Mulint]
```

- bytecode interpreter

```
# interpret [Push; Value 10; Push; Value 20;
             Push; Value 30; Addint; Mulint]
- : int = 500
```

REMARK

use 16/32/64/128-bit integers for instruction in practice

Interpreting Bytecode

	0	1	2	3	4	5	6	7
code = [	Push;	Value 10;	Push;	Value 20;	Push;	Value 30;	Addint;	Mulint]

			pc	stack
interpret code	$=$	interpret′ code (	$0,$	$[\,])$
	$=$	interpret′ code (	$2,$	$10 :: [\,])$
	$=$	interpret′ code (	$4,$	$20 :: 10 :: [\,])$
	$=$	interpret′ code (	$6,$	$30 :: 20 :: 10 :: [\,])$
	$=$	interpret′ code (	$7,$	$50 :: 10 :: [\,])$
	$=$	interpret′ code (	$8,$	$500 :: [\,])$
	$=$	500		

Compiling Expressions

compile (Mul (Const 10, Add (Const 20, Const30)))
= compile (Const 10) @
compile (Add (Const 20, Const 30)) @
[Mulint]
= [Push; Value 10] @
compile (Add (Const 20, Const 30)) @
[Mulint]
= ...
= [Push; Value 10;
Push; Value 20;
Push; Value 30;
Addint;
Mulint]