CS251 Programming Languages **Handout # 13**
Prof. Lyn Turbak **February 6, 2004**
Wellesley College ***Revised February 7,2004***

Problem Set 2
Due: 6pm Thursday, February 12

Revisions:
Feb 7: (1) In `length` example in Problem 2, (a) missing `rec` and (b) second `=` should be `->`.
(2) In Problem 3, `#load "load-ps2-funs.ml"` should be `#use "load-ps2-funs.ml"`.
(3) In `ps2-funs.ml`, missing `nss` in stub for `all_contain_multiple`.

Overview:

The purpose of this assignment is to give you experience with first-class functions. These can twist your brain a bit, so leave sufficient time to do the problems.

Reading:

- Handout #14: First-Class Functions

Working Together:

Reminder: if you worked with a partner on PS1 and want to work with a partner on this assignment, you must choose a different partner.

Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 6pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your solution to Problem 1 (which may be pencil-and-paper).
3. your solution to Problem 2 (which may be pencil-and-paper).
4. your final version of `ps2-funs.ml` for Problem 3
5. your final version of `church.ml` for Problem 4.

Each team should also submit a single softcopy (consisting of your final `ps2` directory) to the drop directory `~cs251/drop/ps2/`*username*, where *username* is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/username/cs251
cp -R ps2 ~cs251/drop/ps2/username/
```

Problem 1 [20]: Substitution Model

Consider the following OCAML declarations:

```
let sub x y = x - y

let app5 f = f 5

let flip f a b = f b a
```

The substitution model introduced in lecture is able to show the step-by-step evaluation of OCAML expressions involving higher-order functions like `app5` and `flip`. Here are two examples illustrating the substitution model (where we have delayed expanding the definitions of top-level variables until we need them):

```
# flip sub 3 2
⇒ (fun f a b -> f b a) sub 3 2
⇒ sub 2 3
⇒ (fun x y -> x - y) 2 3
⇒ 2 - 3
⇒ -1

# flip flip 3 sub 5
⇒ (fun f a b -> f b a) flip 3 sub 5
⇒ flip sub 3 5
⇒ (fun f a b -> f b a) sub 3 5
⇒ sub 5 3
⇒ (fun x y -> x - y) 5 3
⇒ 5 - 3
⇒ 2
```

Note that function application in OCAML is left-associative. For example, `flip sub 3 2` is parsed as `((flip sub) 3) 2` and `flip flip 3 sub 5` is parsed as `(((flip flip) 3) sub) 2` The left associativity of function application dovetails nicely with the right associativity of the arrow type `->` (think about this).

Use the substitution model to show the step-by-step evaluation of the following expressions. You may use OCAML to check your answers, but please do not do so until you have already tried to figure out the answers on your own.

a. [**5**] `app5 (sub 1)`

b. [**5**] `app5 sub 2`

c. [**5**] `app5 (flip sub) 3`

d. [**5**] `flip app5 4 sub`

Problem 2 [20]: Function Types

Figs. 1–2 contain twenty higher-order OCAML functions. For each function, write down the type that would be automatically reconstructed for it. For example, consider the following OCAML `length` function:

```
let rec length xs =
  match xs with
    [] -> 0
  | (_::xs') -> 1 + (length xs')
```

The type of this OCAML function is:

```
'a list -> int
```

Note: you can check your answers by typing them into the OCAML interpreter. But please write down the answers first before you check them — otherwise you will not learn anything!

```
let id x = x

let compose f g x = (f (g x))

let rec repeated n f =
      if (n = 0) then id else compose f (repeated (n - 1) f)

let uncurry f (a,b) = (f a b)

let curry f a b = f(a,b)

let chPair x y = fun f -> f x y

let rec gen next isDone seed =
  if (isDone seed) then
    []
  else
    seed :: (gen next isDone (next seed))

let rec map f xs =
  match xs with
    [] -> []
  | (x::xs') -> (f x) :: (map f xs')

let rec filter pred xs =
  match xs with
    [] -> []
  | (x::xs') ->
      if (pred x) then
        x::(filter pred xs')
      else
        filter pred xs'

let product fs xs =
  map (fun f -> map (fun x -> (f x)) xs) fs
```

Figure 1: A sampler of higher-order functions in OCAML, part 1.

```
let rec zip pair =
  match pair with
    ([], _) -> []
  | (_, []) -> []
  | (x::xs', y::ys') -> (x,y)::(zip(xs',ys'))

let rec unzip xys =
  match xys with
    [] -> ([], [])
  | ((x,y)::xys') ->
      let (xs,ys) = unzip xys'
       in (x::xs, y::ys)

let rec foldr binop init xs =
  match xs with
    [] -> init
  | (x::xs) -> binop x (foldr binop init xs)

let foldr2 ternop init xs ys =
  foldr (fun (x,y) ans -> (ternop x y ans)) init (zip(xs,ys))

let flatten xss = foldr (@) [] xss

let rec forall pred xs =
  match xs with
    [] -> true
  | (x::xs') -> pred(x) && (forall pred xs')

let rec exists pred xs =
  match xs with
    [] -> false
  | (x::xs') -> (pred x) || (exists pred xs')

let rec some pred xs =
  match xs with
    [] -> None
  | (x::xs') -> if (pred x) then Some x else some pred xs'

let oneListOpToTwoListOp f =
  let twoListOp binop xs ys = f binop (zip(xs,ys))
   in twoListOp

let some2 pred = oneListOpToTwoListOp some pred
```

Figure 2: A sampler of higher-order functions in OCAML, part 2.

Problem 3 [45]: Higher-Order List Functions

Below you are asked to write several function declarations, most of which are curried versions of functions you implemented in Problem Set 1. The difference is that here you are not allowed to use explicit recursion to solve any of the problems. Instead, you should use the (mostly higher-order) list functions in `utils/ListUtils.ml`. A few other handy functions are defined in `utils/FunUtils.ml`. All definitions can be written in a few lines, and most can be written in one line.

Stubs for all the functions can be found in `ps2/ps2-funs.ml`. You should flesh out the definitions in this file. To try out your functions, execute the following in the OCAML interpreter:

```
#cd "/students/username/cs251/ps2"
#use "load-ps2-funs.ml"
```

This will load `FunUtils.ml` as well as `ListUtils.ml`.

The file `ps2-funs.ml` begins with the declarations:

```
open FunUtils
open ListUtils
```

This makes all functions in the `FunUtils` and `ListUtils` modules available in `funs.ml` without the need for explicit qualification. E.g., you can write `id` rather than `FunUtils.id` and `map` rather than `List.map`.

a. [**5**] `val sum_multiples_of_3_or_5 : int * int -> int`

`sum_multiples_of_3_or_5` *`m n`* returns the sum of all integers from m up to n (inclusive) that are multiples of 3 and/or 5. For example:

```
# sum_multiples_of_3_or_5 0 10;;
- : int = 33 (* 3 + 5 + 6 + 9 + 10 *)
# sum_multiples_of_3_or_5 (-9) 12;;
- : int = 22
# sum_multiples_of_3_or_5 18 18;;
- : int = 18
# sum_multiples_of_3_or_5 10 0;;
- : int = 0 (* The range "10 up to 0" is empty. *)
```

b. [**5**] `val all_contain_multiple : int -> int list list -> bool`

`all_contain_multiple` *`n nss`* returns `true` if each list of integers in `nss` contains at least one integer that is a multiple of n; otherwise it returns `false`.

```
# all_contain_multiple 5 [[17;10;12]; [25]; [3;7;5]];;
- : bool = true
# all_contain_multiple 3 [[17;10;12]; [25]; [3;7;5]];;
- : bool = false
# all_contain_multiple 3 [];;
- : bool = true
```

c. [**5**] `val inner_product :  int list -> int list -> int`

Assume that *xs* is the list of integers $[x_1, \ldots, x_n]$ and *ys* is the list of integers $[y_1, \ldots, y_n]$. (Both lists are assumed to have the same length n.) Returns $\sum_{i=1}^{n} x_i \cdot y_i$.

```
# inner_product [1;2;3] [4;5;6];;
- : int = 32 (* 4 + 10 + 18 *)
# inner_product [] [];;
- : int = 0
```

d. [5] `val alts : 'a list -> 'a list * 'a list`

Assume that the elements of a list are indexed starting with 1. `alts` *xs* returns a pair of lists, the first of which has all the odd-indexed elements (in the same relative order as in *xs*) and the second of which has all the even-indexed elements (in the same relative order as in *xs*).

```
# alts [7;5;4;6;9;2;8;3];;
- : int list * int list = ([7; 4; 9; 8], [5; 6; 2; 3])
# alts [7;5;4;6;9;2;8];;
- : int list * int list = ([7; 4; 9; 8], [5; 6; 2])
# alts [7];;
- : int list * int list = ([7], [])
# alts [];;
- : '_a list * '_a list = ([], [])
```

e. [5] `val cartesian_product : 'a list -> 'b list -> ('a * 'b) list`

`cartesian_product` *xs ys* returns a list of all pairs `(`*x*`,`*y*`)` where *x* ranges over the elements of *xs* and *y* ranges over the elements of *ys*. The pairs should be sorted first by the *x* entry (relative to the order in *xs*) and then by the *y* entry (relative to the order in *ys*).

```
# cartesian_product [1; 2] ['a'; 'b'; 'c'];;
- : (int * char) list =
[(1, 'a'); (1, 'b'); (1, 'c'); (2, 'a'); (2, 'b'); (2, 'c')]
# cartesian_product [2; 1] ['a'; 'b'; 'c'];;
- : (int * char) list =
[(2, 'a'); (2, 'b'); (2, 'c'); (1, 'a'); (1, 'b'); (1, 'c')]
# cartesian_product ['c'; 'a'; 'b'] [2; 1];;
- : (char * int) list =
[('c', 2); ('c', 1); ('a', 2); ('a', 1); ('b', 2); ('b', 1)]
# cartesian_product [1] ['a'];;
- : (int * char) list = [(1, 'a')]
# cartesian_product [] ['a'; 'b'; 'c'];;
- : ('_a * char) list = []
```

f. [5] `val bits : int -> int list`

`bits` *n* returns a list of the bits (0s and 1s) in the binary representation of *n*.

```
# bits 5;;
- : int list = [1; 0; 1]
# bits 10;;
- : int list = [1; 0; 1; 0]
# bits 11;;
- : int list = [1; 0; 1; 1]
# bits 22;;
- : int list = [1; 0; 1; 1; 0]
# bits 23;;
- : int list = [1; 0; 1; 1; 1]
# bits 46;;
- : int list = [1; 0; 1; 1; 1; 0]
# bits 0;;
- : int list = [0] (* special case! *)
```

g. [**5**] `val n_fold : int -> ('a -> 'a) -> 'a -> 'a`

`n_fold` *`n f`* returns the *n*-fold composition of the function *f*.

```
# n_fold 5 ((+) 1) 0;;
- : int = 5
# n_fold 5 (( * ) 2) 1;;
- : int = 32
# n_fold 3 ((flip (/)) 2) 100;;
- : int = 12
# n_fold 0 ((flip (/)) 2) 100;;
- : int = 100
```

h. [**5**] `val inserts : 'a -> 'a list -> 'a list list`

Assume that *ys* is a list with *n* elements. `insert (`*`x`*`,`*`ys`*`)` returns a $n+1$-length list of lists showing all ways to insert a single copy of *x* into *xs*.

```
# inserts 3 [5;7;1];;
- : int list list = [[3; 5; 7; 1]; [5; 3; 7; 1]; [5; 7; 3; 1]; [5; 7; 1; 3]]
# inserts 3 [5;3;1];;
- : int list list = [[3; 5; 3; 1]; [5; 3; 3; 1]; [5; 3; 3; 1]; [5; 3; 1; 3]]
# inserts 3 [];;
- : int list list = [[3]]
```

Hint: It is possible to solve this problem with `foldr`, but it is easier to use `foldr'`.

i. [**5**] `val permutations : 'a list -> 'a list list`

Assume that *xs* is a list of distinct elements (i.e., no duplicates). `permutations` *`xs`* returns a list of all the permutations of the elements of *xs*. The order of the permutations does not matter.

```
# permutations [];;
- : '_a list list = [[]]
# permutations [1];;
- : int list list = [[1]]
# permutations [1;2];;
- : int list list = [[1; 2]; [2; 1]]
# permutations [1;2;3];;
- : int list list =
[[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
# permutations [1;2;3;4];;
- : int list list =
[[1; 2; 3; 4]; [2; 1; 3; 4]; [2; 3; 1; 4]; [2; 3; 4; 1]; [1; 3; 2; 4];
 [3; 1; 2; 4]; [3; 2; 1; 4]; [3; 2; 4; 1]; [1; 3; 4; 2]; [3; 1; 4; 2];
 [3; 4; 1; 2]; [3; 4; 2; 1]; [1; 2; 4; 3]; [2; 1; 4; 3]; [2; 4; 1; 3];
 [2; 4; 3; 1]; [1; 4; 2; 3]; [4; 1; 2; 3]; [4; 2; 1; 3]; [4; 2; 3; 1];
 [1; 4; 3; 2]; [4; 1; 3; 2]; [4; 3; 1; 2]; [4; 3; 2; 1]]
```

Problem 4 [15]: Church Numerals

The First-Class Functions handout (#14) discusses how n-fold composition functions (so-called Church numerals) can be viewed as the basis of a system for arithmetic. Write OCAML definitions for the functions `times`, `expt`, and `pred` that that are described in the handout. Flesh out these definitions in the file `~/cs251/ps2/church.ml`, which includes other definitions from Handout #14. Evaluate `#use "load-church.ml"` to load the file into ocaml.

Notes:

- Your definitions should not use any of the following: `n_fold`, `int2ch`, `ch2int`, or any recursively defined function.
- Your definitions may use any other functions. In particular, the following functions are useful: `id`, `o` (compose), `nonce`, `once`, `succ`, `plus`, `chPair`, `chFst`, and `chSnd`. Although some solution strategies use some of these functions, none are absolutely required. Indeed, there are solutions for all three definitions that use none of the above functions.
- For ideas on how to implement these three functions, carefully study the definitions of `succ`, `plus`, and `plus'` as well as the examples involving `twice` and `thrice` in the function composition section of the First-Class Functions handout. You can implement `times` and `expt` by generalizing patterns you see in invocations of `twice` and `thrice`. The `pred` function is very challenging. *Hint:* One approach is to use an iteration on pairs.
- Each of your function definitions should be extremely short. It is possible to implement each definition as a "one-liner".

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS251 Problem Set 2
Due 6pm Thursday, February 12

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with*):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.
Signature(s):

In the **Time** *column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading your problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [20]		
Problem 2 [20]		
Problem 3 [45]		
Problem 4 [15]		
Total		