

Name:

Hemos ID:

CSE-321 Programming Languages 2010
Midterm — Sample Solution

	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score					
Max	15	30	35	20	100

1 SML Programming [15 pts]

Question 1. [5 pts] Give a tail recursive implementation of `preorder` for preorder traversals of binary trees. Fill in the blank:

```
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree

(* preorder : 'a tree -> 'a list *)
fun preorder t =
  let

    fun preorder' (Leaf x) pre =
      _____

      pre @ [x]
      _____

    | preorder' (Node (left, x, right)) pre =
      _____

      preorder' right (preorder' left (pre @ [x]))
      _____
  in
    preorder' t []
  end
```

Question 2. [7 pts] `DICT` is a signature for dictionaries:

```
signature DICT =
sig
  type key
  type 'a dict
  val empty : unit -> 'a dict
  val lookup : 'a dict -> key -> 'a option
  val delete : 'a dict -> key -> 'a dict
  val insert : 'a dict -> key * 'a -> 'a dict
end
```

- `key` denotes the type of keys in dictionaries.
- `'a dict` denotes the type of dictionaries for `'a` type values.
- `empty ()` returns an empty dictionary.
- `lookup d k` searches the key `k` in the dictionary `d`. If the key is found, it returns the associated item. Otherwise, it returns `NONE`.
- `delete d k` deletes the key `k` and its associated item in the dictionary `d` and returns the resultant dictionary `d'`. If the key does not exist in the dictionary `d`, it returns the given dictionary `d` without any modification.
- `insert d (k, v)` inserts the new key `k` and its associated item `v` in the dictionary `d`. If the key `k` already exists in the dictionary `d`, it just updates its associated item with the given item `v`.

Implement the functor DictFn which takes a KEY structure and generates a corresponding DICT structure that uses a ‘functional representation’ of dictionaries. Fill in the blank:

```
signature KEY =
sig
  type t

  (* eq k k' : true    k is equal to k'  *)
  (*          false   otherwise          *)
  val eq : t * t -> bool
end

functor DictFn (Key : KEY) :> DICT where type key = Key.t
=
struct
  type key = Key.t
  type 'a dict = key -> 'a option

  fun empty () =

    fn _ => NONE

  fun lookup d k =

    d k

  fun delete d k =

    fn k' => if Key.eq (k, k') then NONE else d k'

  fun insert d (k, v) =

    fn k' => if Key.eq (k, k') then SOME v else d k'
end
```

Question 3. [3 pts] Give an implementation of `IntDict` whose key type is `int`. Fill in the blank. You may use the functor `DictFn` that you write in Question 2.

```
structure IntKey :> KEY where type t = int
=
struct
```

```
  type t = int
```

```
  fun eq (k, k') = k = k'
```

```
end
```

```
structure IntDict = DictFn (IntKey)
```

2 Inductive definitions [30 pts]

Consider the following system from the Course Notes where s `mparen` means that s is a string of matched parentheses.

$$\frac{}{\epsilon \text{ mparen}} \text{Meps} \quad \frac{s \text{ mparen}}{(s) \text{ mparen}} \text{Mpar} \quad \frac{s_1 \text{ mparen} \quad s_2 \text{ mparen}}{s_1 s_2 \text{ mparen}} \text{Mseq}$$

In order to show that if s `mparen` holds, s is indeed a string of matched parentheses, we introduce a new judgment $k \triangleright s$ where k is a non-negative integer:

$$\begin{aligned} k \triangleright s &\Leftrightarrow k \text{ left parentheses concatenated with } s \text{ form a string of matched parentheses} \\ &\Leftrightarrow \underbrace{((\dots (}_{k} s \text{ is a string of matched parentheses} \end{aligned}$$

The idea is that we scan a given string from left to right and keep counting the number of left parentheses that have not yet been matched with corresponding right parentheses. Thus we begin with $k = 0$, increment k each time a left parenthesis is encountered, and decrement k each time a right parenthesis is encountered:

$$\frac{}{0 \triangleright \epsilon} \text{Peps} \quad \frac{k+1 \triangleright s}{k \triangleright (s)} \text{Pleft} \quad \frac{k-1 \triangleright s \quad k > 0}{k \triangleright)s} \text{Prigh}$$

The second premise $k > 0$ in the rule *Prigh* ensures that in any prefix of a given string, the number of right parentheses may not exceed the number of left parentheses. Now a judgment $0 \triangleright s$ expresses that s is a string of matched parentheses.

Your task is to prove Theorem 2.1. If you need a lemma to complete the proof, state the lemma, prove it, and use it to complete the proof of Theorem 2.1.

For individual steps in the proof, please use the following format:

conclusion

justification

Theorem 2.1. *If s `mparen`, then $0 \triangleright s$.*

We first introduce the following lemma:

Lemma 2.2. *If $k_1 \triangleright s_1$, then $k_2 \triangleright s_2$ implies $k_1 + k_2 \triangleright s_1 \ s_2$.*

Proof. By rule induction on $k_1 \triangleright s_1$.

Case $\frac{}{0 \triangleright \epsilon} Peps$ where $k_1 = 0$ and $s_1 = \epsilon$:

$\frac{k_1 + k_2 = k_2}{}$	$\frac{\text{from } k_1 = 0}{}$
$\frac{s_1 \ s_2 = s_2}{}$	$\frac{\text{from } s_1 = \epsilon}{}$
$\frac{k_1 + k_2 \triangleright s_1 \ s_2}{}$	$\frac{\text{assumption}}{}$

Case $\frac{k_1 + 1 \triangleright s}{k_1 \triangleright (s)} Pleft$ where $s_1 = (s$:

$\frac{k_1 + 1 + k_2 \triangleright s \ s_2}{}$	$\frac{\text{by IH on } k_1 + 1 \triangleright s}{}$
$\frac{\frac{k_1 + 1 + k_2 \triangleright s \ s_2}{k_1 + k_2 \triangleright (s \ s_2)} Pleft}{}$	

Case $\frac{k_1 - 1 \triangleright s \ k_1 > 0}{k_1 \triangleright)s} Pright$ where $s_1 =)s$:

$\frac{k_1 - 1 + k_2 \triangleright s \ s_2}{}$	$\frac{\text{by IH on } k_1 - 1 \triangleright s}{}$
$\frac{k_1 + k_2 > 0}{}$	$\frac{\text{from } k_1 > 0 \text{ and } k_2 \geq 0}{}$
$\frac{\frac{k_1 - 1 + k_2 \triangleright s \ s_2 \ k_1 + k_2 > 0}{k_1 + k_2 \triangleright)s \ s_2} Pright}{}$	

□

Proof of Theorem 2.1. By rule induction on s mparen.

Case $\overline{\epsilon \text{ mparen}} \text{ Meps}$ where $s = \epsilon$:

$\overline{0 \triangleright \epsilon}$	by the rule <i>Peps</i>
--	-------------------------

Case $\frac{s' \text{ mparen}}{(s') \text{ mparen}} \text{ Mpar}$ where $s = (s')$:

$\overline{0 \triangleright s'}$	by IH on $s' \text{ mparen}$
----------------------------------	------------------------------

$\frac{\overline{0 \triangleright \epsilon} \text{ Peps} \quad 1 > 0 \text{ Pright}}{1 \triangleright)}$

$\overline{1 \triangleright s')}$	by Lemma 2.2 with $0 \triangleright s'$ and $1 \triangleright)$
------------------------------------	--

$\overline{0 \triangleright (s')}$	by the rule <i>Pleft</i>
------------------------------------	--------------------------

Case $\frac{s_1 \text{ mparen} \quad s_2 \text{ mparen}}{s_1 s_2 \text{ mparen}} \text{ Mseq}$ where $s = s_1 s_2$:

$\overline{0 \triangleright s_1}$	by IH on $s_1 \text{ mparen}$
-----------------------------------	-------------------------------

$\overline{0 \triangleright s_2}$	by IH on $s_2 \text{ mparen}$
-----------------------------------	-------------------------------

$\overline{0 \triangleright s_1 s_2}$	by Lemma 2.2
---------------------------------------	--------------

□

3 λ -Calculus [35 pts]

Question 1. [5 pts] Show the reduction sequence under the call-by-name strategy. Underline the redex at each step.

$$(\lambda x. \lambda y. y \ x) ((\lambda x. x) (\lambda y. y)) (\lambda z. z)$$

$$\mapsto \underline{(\lambda y. y ((\lambda x. x) (\lambda y. y))) (\lambda z. z)}$$

$$\mapsto \underline{(\lambda z. z) ((\lambda x. x) (\lambda y. y))}$$

$$\mapsto \underline{(\lambda x. x) (\lambda y. y)}$$

$$\mapsto \underline{(\lambda y. y)}$$

Question 2. [3 pts] Complete the definition of $FV(e)$ that finds the set of free variables in e .

$$FV(x) = \underline{\{x\}}$$

$$FV(\lambda x. e) = \underline{FV(e) - \{x\}}$$

$$FV(e_1 \ e_2) = \underline{FV(e_1) \cup FV(e_2)}$$

Question 3. [2 pts] Fill in the blank with the set of free variables of the given expression.

$$FV(\lambda x. x) = \underline{\{\}} \quad \underline{\hspace{2cm}}$$

$$FV(x \ y) = \underline{\{x, y\}} \quad \underline{\hspace{2cm}}$$

$$FV(\lambda x. x \ y) = \underline{\{y\}} \quad \underline{\hspace{2cm}}$$

$$FV(\lambda x. \lambda y. x \ y) = \underline{\{\}} \quad \underline{\hspace{2cm}}$$

$$FV((\lambda x. x \ y) (\lambda y. x \ y)) = \underline{\{y, x\}} \quad \underline{\hspace{2cm}}$$

Question 4. [5 pts] This question assumes types `var` and `exp` that we have seen in Assignment 4:

```
type var = string
datatype exp =
  Var of var
| Lam of var * exp
| App of exp * exp
```

Suppose that we have two functions `notFv` and `varSwap`:

- `notFv : var -> exp -> bool`
`notFv x e` returns true if x is a free variable of e and false otherwise.
- `varSwap : var * var -> exp -> exp`
`varSwap (x, y) e` returns $[x \leftrightarrow y]e$.

Below is a function `aEqual` of type `(exp * exp) -> bool` such that `aEqual (e1, e2)` returns true if e_1 and e_2 are α -equivalent and false otherwise.

```
fun aEqual (Var x, Var y) = x = y
| aEqual (App (e1, e2), App (e1', e2')) =
  aEqual (e1, e1') andalso aEqual (e2, e2')
| aEqual (Lam (x, e), Lam (y, e')) =
  if x = y then aEqual (e, e')
  else if notFv x e' then aEqual (e, varSwap (y, x) e')
  else false
| aEqual _ = false
```

We write $e \equiv_\alpha e'$ if e can be rewritten as e' by renaming bound variables in e and vice versa. Give exactly four inference rules corresponding to the above definition of `aEqual`. Use the notation $x \notin FV(e)$ for `notFv x e` and $[x \leftrightarrow y]e$ for `varSwap (x, y) e`.

$$\frac{}{x \equiv_\alpha x}$$

$$\frac{e_1 \equiv_\alpha e'_1 \quad e_2 \equiv_\alpha e'_2}{e_1 e_2 \equiv_\alpha e'_1 e'_2}$$

$$\frac{e \equiv_\alpha e'}{\lambda x. e \equiv_\alpha \lambda x. e'}$$

$$\frac{x \neq y \quad x \notin FV(e') \quad e \equiv_\alpha [y \leftrightarrow x]e'}{\lambda x. e \equiv_\alpha \lambda y. e'}$$

Question 5. [8 pts] A Church numeral encodes a natural number n as a λ -abstraction \hat{n} which takes a function f and returns $f^n = f \circ f \cdots \circ f$ (n times):

$$\hat{n} = \lambda f. f^n = \lambda f. \lambda x. f \ f \ f \ \cdots \ f \ x$$

In this question, you will define three functions: **sub** for the subtraction operation, **mod** for the modulo operation, and, as an extra credit problem, **div** for the division operation.

Your answers may use the following pre-defined constructs: **zero**, **one**, **succ**, **if/then/else**, **pair/fst/snd**, **pred**, **eq**, and **fix**.

- **zero** and **one** encode the natural numbers zero and one, respectively.

$$\begin{aligned} \text{zero} &= \hat{0} = \lambda f. \lambda x. x \\ \text{one} &= \hat{1} = \lambda f. \lambda x. f \ x \end{aligned}$$

- **succ** finds the successor of a given natural number.

$$\text{succ} = \lambda \hat{n}. \lambda f. \lambda x. \hat{n} \ f \ (f \ x)$$

- **if e then e_1 else e_2** is a conditional construct.

$$\text{if } e \text{ then } e_1 \text{ else } e_2 = e \ e_1 \ e_2$$

- **pair** creates a pair of two expressions, and **fst** and **snd** are projection operators.

$$\begin{aligned} \text{pair} &= \lambda x. \lambda y. \lambda b. b \ x \ y \\ \text{fst} &= \lambda p. p \ (\lambda t. \lambda f. t) \\ \text{snd} &= \lambda p. p \ (\lambda t. \lambda f. f) \end{aligned}$$

- **pred** computes the predecessor of a given natural number where the predecessor of 0 is 0.

$$\text{pred} = \lambda \hat{n}. \text{fst} \ (\hat{n} \ \text{next} \ (\text{pair} \ \text{zero} \ \text{zero}))$$

- **eq** tests two natural numbers for equality.

$$\text{eq} = \lambda x. \lambda y. \text{and} \ (\text{isZero} \ (x \ \text{pred} \ y)) \ (\text{isZero} \ (y \ \text{pred} \ x))$$

- **fix** is the fixed point combinator.

$$\text{fix} = \lambda F. (\lambda f. F \ \lambda x. (f \ f \ x)) \ (\lambda f. F \ \lambda x. (f \ f \ x))$$

These constructs use the following auxiliary constructs, which you do not need:

$$\begin{aligned} \text{tt} &= \lambda t. \lambda f. t \\ \text{ff} &= \lambda t. \lambda f. f \\ \text{and} &= \lambda x. \lambda y. x \ y \ \text{ff} \\ \text{isZero} &= \lambda x. x \ (\lambda y. \text{ff}) \ \text{tt} \\ \text{next} &= \lambda p. \text{pair} \ (\text{snd} \ p) \ (\text{succ} \ (\text{snd} \ p)) \end{aligned}$$

Define a subtraction function **sub** such that **sub** \hat{m} \hat{n} evaluates to $\widehat{m - n}$ if $m > n$ and $\hat{0}$ otherwise.

sub = $\lambda\hat{m}. \lambda\hat{n}. (\hat{n} \text{ pred}) \hat{m}$

Define a modulo function **mod** such that **mod** \hat{m} \hat{n} evaluates to \hat{r} if r is the remainder of division of m by n . **mod** never takes $\hat{0}$ as the second argument. Hence the result of evaluating **mod** \hat{m} $\hat{0}$ is unspecified. You may use the subtraction function **sub** that you define above.

mod = $\text{fix } (\lambda f. \lambda\hat{m}. \lambda\hat{n}.$
 if eq \hat{m} \hat{n} then zero
 else if eq (sub \hat{m} \hat{n}) zero then \hat{m}
 else (f (sub \hat{m} \hat{n}) \hat{n}))

Extra credit question. [10 pts] Define a division function **div** such that **div** \hat{m} \hat{n} evaluates to \hat{q} if q is the quotient of m divided by n . **div** never takes $\hat{0}$ as the second argument. Hence the result of evaluating **div** \hat{m} $\hat{0}$ is unspecified. In this question, you are not allowed to use the fixed point combinator (and its definition), but you may use the subtraction function **sub** that you define above.

div = $\lambda\hat{m}. \lambda\hat{n}. \text{snd } (\hat{m} (\lambda p. \text{if eq (fst } p) \hat{n} \text{ then pair zero (succ (snd } p))$
 else if eq (sub (fst p) \hat{n}) zero then p else pair (sub (fst p) \hat{n}) (succ (snd p)))

 (pair \hat{m} zero))

Question 6. [3 pts] Give an expression whose reduction does not terminate.

$$\underline{(\lambda x. x x) (\lambda x. x x)}$$

Question 7. [5 pts] Following is the definition of de Bruijn expressions:

$$\begin{array}{ll} \text{de Bruijn expression} & M ::= n \mid \lambda. M \mid M M \\ \text{de Bruijn index} & n ::= 0 \mid 1 \mid 2 \mid \dots \end{array}$$

Suppose that you are given the definition of $\tau_i^n(N)$ for shifting by n (*i.e.*, incrementing by n) all de Bruijn indexes in N corresponding to free variables, where a de Bruijn index m in N such that $m < i$ does not count as a free variable.

Complete the definition of $\sigma_n(M, N)$ for substituting N for every occurrence of n in M where N may include free variables.

$$\sigma_n(M_1 M_2, N) = \underline{\sigma_n(M_1, N) \sigma_n(M_2, N)}$$

$$\sigma_n(\lambda. M, N) = \underline{\lambda. \sigma_{n+1}(M, N)}$$

$$\sigma_n(m, N) = \underline{m} \quad \text{if } m < n$$

$$\sigma_n(n, N) = \underline{\tau_0^n(N)}$$

$$\sigma_n(m, N) = \underline{m - 1} \quad \text{if } m > n$$

Question 8. [4 pts] Show the reduction of the given expression where the redex is underlined.

$$\lambda. \lambda. \underline{(\lambda. (\lambda. 3 \ 2 \ 1 \ 0) (2 \ 1 \ 0)) (\lambda. 0)} \mapsto \underline{\lambda. \lambda. (\lambda. 2 \ 1 (\lambda. 0) 0) (1 \ 0 (\lambda. 0))}$$

$$\underline{(\lambda. (\lambda. 1) 0) (\lambda. 2 \ 1 \ 0)} \mapsto \underline{(\lambda. \lambda. 3 \ 2 \ 0) (\lambda. 2 \ 1 \ 0)}$$

4 Simply-typed λ -calculus [20 pts]

Question 1. [2 pts] Consider the following simply-typed λ -calculus:

$$\begin{array}{ll} \text{type} & A ::= \text{bool} \mid A \rightarrow A \\ \text{expression} & e ::= x \mid \lambda x:A. e \mid e e \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \end{array}$$

Write the typing rules for x , $\lambda x:A. e$, and $e e$:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x:A. e : A \rightarrow B}$$

$$\frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e e' : B}$$

Question 2. [3 pts] Consider the extension of the simply-typed λ -calculus with product types:

$$\begin{array}{ll} \text{type} & A ::= \dots \mid A \times A \\ \text{expression} & e ::= \dots \mid (e, e) \mid \text{fst } e \mid \text{snd } e \end{array}$$

Write the reduction rules for these constructs under *lazy* reduction strategy:

$$\frac{e \mapsto e'}{\text{fst } e \mapsto \text{fst } e'}$$

$$\overline{\text{fst } (e_1, e_2) \mapsto e_1}$$

$$\frac{e \mapsto e'}{\text{snd } e \mapsto \text{snd } e'}$$

$$\overline{\text{snd } (e_1, e_2) \mapsto e_2}$$

Question 3. [5 pts] Consider the extension of the simply-typed λ -calculus with sum types:

$$\begin{array}{ll} \text{type} & A ::= \dots \mid A + A \\ \text{expression} & e ::= \dots \mid \text{inl}_A e \mid \text{inr}_A e \mid \text{case } e \text{ of } \text{inl } x. e \mid \text{inr } x. e \end{array}$$

Write the typing rules:

$$\frac{\Gamma \vdash e : B}{\Gamma \vdash \text{inl}_A e : B + A}$$

$$\frac{\Gamma \vdash e : B}{\Gamma \vdash \text{inr}_A e : A + B}$$

$$\frac{\Gamma \vdash e : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : C \quad \Gamma, x_2 : A_2 \vdash e_2 : C}{\Gamma \vdash \text{case } e \text{ of } \text{inl } x_1. e_1 \mid \text{inr } x_2. e_2 : C}$$

Question 4. [5 pts] Consider the extension of the simply-typed λ -calculus with fixed point constructs

$$\text{expression } e ::= \dots \mid \text{fix } x : A. e$$

Write the typing rule for $\text{fix } x : A. e$ and its reduction rule.

$$\frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash \text{fix } x : A. e : A}$$

$$\overline{\text{fix } x : A. e \mapsto [\text{fix } x : A. e / x]e}$$

Question 5. [5 pts] Consider the following SML program:

```
fun even 0 = true
  | even 1 = false
  | even n = odd (n - 1)

and odd 0 = false
  | odd 1 = true
  | odd n = even (n - 1)
```

The function `even` calls the function `odd`, and the function `odd` calls the function `even`. We refer to these functions as mutually recursive functions.

Write an expression of type $(\text{int} \rightarrow \text{bool}) \times (\text{int} \rightarrow \text{bool})$ that encodes both `even` and `odd` in the simply-typed λ -calculus:

type	$A ::= \text{int} \mid \text{bool} \mid A \rightarrow A \mid A \times A$
expression	$e ::= x \mid \lambda x:A. e \mid e e \mid (e, e) \mid \text{fst } e \mid \text{snd } e \mid () \mid$ $\text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \text{fix } x:A. e$ $- \mid = \mid 0 \mid 1 \mid \dots$

We assume that the infix operations $-$ and $=$ are given as primitive, which correspond to the integer substitution and equality test, respectively.

`fix f : ((int \rightarrow bool) \times (int \rightarrow bool)).`

`(λn : int.`

`if $n = 0$ then true else`

`if $n = 1$ then false else`

`(snd f) ($n - 1$),`

`λn : int.`

`if $n = 0$ then false else`

`if $n = 1$ then true else`

`(fst f) ($n - 1$))`