Name: Hemos ID:

CSE-321 Programming Languages 2015
Midterm

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Total
Score							
Max	10	20	20	15	15	20	100

- There are six problems on 12 pages in this exam.
- The maximum score for this exam is 100 points.
- Be sure to write your name and Hemos ID.
- For writing code in Problems 1 and 3, write your answers exactly as you would type on the screen. See the instruction given in Problem 1.
- When writing individual proof steps, please write *conclusion* in the left and *justification* in the right, as in the course notes.
- Write your proof legibly.
- You have three hours for this exam.

1 OCaml Programming [10 points]

In this problem, you will implement functions satisfying given descriptions. You should write one character per blank. For example, the following code implements a sum function.

```
l e t _ r e c _ s u m _ n _ = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ i f _ n _ = _ 0 _ t h e n _ 0 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ e l s e _ n _ + _ s u m _ ( n _ - _ 1 ) _ _ _ _ _ _ _ _ _
```

Question 1. [5 points] Assume a recursive datatype `tree` defined as follows:

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

Give a *tail-recursive* implementation of `inorder` for an inorder traversal of binary trees.

(Type) `inorder: 'a tree -> 'a list`

(Description) `inorder` t returns a list of elements produced by an inorder traversal of the tree t.

(Example)

```
# inorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4));;
- : int list = [1; 3; 2; 7; 4]
```

```
let inorder t =
```

Question 2. **[5 points]** Complete the function `lrev` using `List.fold_right` which has the following type:

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

You may use the operator `@` for list concatenation.

(Type) `lrev: 'a list -> 'a list`

(Description) `lrev` l reverses l.

(Example)

```
# lrev [1; 2; 3; 4];;
- : int list = [4; 3; 2; 1]
# lrev [];;
- : 'a list = []
```

```
let lrev l =
```

2 Inductive proof on strings of matched parentheses [20 points]

Consider the following system from the course notes where s mparen means that s is a string of matched parentheses.

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

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

$$\begin{array}{lcl} k \rhd s & \Leftrightarrow & k \textit{ left parentheses concatenated with } s \textit{ form a string of matched parentheses} \\ & \Leftrightarrow & \underbrace{((\cdots(}_{k}\, s \textit{ is a string of matched parentheses} \end{array}$$

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:

$$\frac{}{0 \rhd \epsilon}\ \mathit{Peps} \qquad \frac{k+1 \rhd s}{k \rhd (s}\ \mathit{Pleft} \qquad \frac{k-1 \rhd s \quad k > 0}{k \rhd)s}\ \mathit{Pright}$$

The second premise $k > 0$ in the rule *Pright* 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 $0 \rhd s$ expresses that s is a string of matched parentheses.

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* $0 \rhd s$.

3 Untyped λ-Calculus [20 points]

The abstract syntax of the untyped λ-calculus is given as follow:

$$\text{expression} \quad e \quad ::= \quad x \mid \lambda x.\, e \mid e\ e$$

We may use other names for variables (*e.g.*, z, s, t, f, *arg*, *accum*, and so on). The scope of a λ-abstraction $\lambda x.\, e$ extends as far to the right as possible. We use a reduction judgment of the form $e \mapsto e'$:

$$e \mapsto e' \qquad \Leftrightarrow \qquad e \textit{ reduces to } e'$$

Question 1. [3 points] Give reduction rules for the call-by-name strategy. You may use $[e'/x]e$ for the capture-avoiding substitution of e' for every occurrence of x in e.

Question 2. [4 points] We use a judgment $e \equiv_\alpha e'$ to mean that e can be rewritten as e' by renaming bound variables in e and vice versa. Give four inference rules for the definition of $e \equiv_\alpha e'$. You may use $FV(e)$ for the set of free variables in e and $[x \leftrightarrow y]e$ for variable swapping which replaces *all* occurrences of x in e by y and *all* occurrences of y in e by x.

Question 3. [8 points] Consider the following definitions in OCaml:

```
exception Stuck

type var = string

type exp =
    Var of var
  | Lam of var * exp
  | App of exp * exp

let rec subst e e' x = (* omitted *)
```

The datatype `exp` corresponds to the syntactic category expression in the course notes:

- `Var` x denotes a variable x.
- `Lam (x, e)` denotes a λ-abstraction $\lambda x.\, e$.
- `App (`e_1`,` e_2`)` denotes an application $e_1\ e_2$.

`subst` implements the capture-avoiding substitution in the untyped λ-calculus:

- `subst` has type `exp -> exp -> var -> exp`.
- `subst` e e' x represents $[e'/x]e$, *i.e.*, the capture-avoiding substitution of e' for every occurrence of x in e.

Implement a function `stepv` for one-step reduction in the untyped λ-calculus:

```
val stepv : exp -> exp
```

- `stepv` takes an expression e of type `exp` and returns another expression e' such that $e \mapsto e'$.
- If there is no such expression e', an exception `Stuck` is raised.
- `stepv` uses the call-by-value strategy.
- `stepv` uses `raise Stuck` exactly once. In order to meet this requirement, your implementation may use the wildcard pattern `_` in OCaml.
- Your implementation should *not* introduce auxiliary functions in the body of `stepv`.
- Your implementation should *not* use nested pattern matching, *i.e.* patter matching insider another pattern matching.

```
let rec stepv e =
```

Question 4. [5 points] 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):

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

Define a function halve which halves a given natural number (encoded as a Church numeral):

- halve $\widehat{2 * k}$ returns $\widehat{k}$.
- halve $\widehat{2 * k + 1}$ returns $\widehat{k}$.

You may use the following pre-defined constructs: zero, succ, and pair/fst/snd.

- zero encodes the natural number zero.

$$\mathsf{zero} \quad = \quad \hat{0} \ = \ \lambda f.\, \lambda x.\, x$$

- succ finds the successor of a given natural number.

$$\mathsf{succ} \quad = \quad \lambda \widehat{n}.\, \lambda f.\, \lambda x.\, \widehat{n}\ f\ (f\ x)$$

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

$$\begin{array}{rcl} \mathsf{pair} & = & \lambda x.\, \lambda y.\, \lambda b.\, b\ x\ y \\ \mathsf{fst} & = & \lambda p.\, p\ (\lambda t.\, \lambda f.\, t) \\ \mathsf{snd} & = & \lambda p.\, p\ (\lambda t.\, \lambda f.\, f) \end{array}$$

$\mathsf{halve} \quad =$ _____

4 De Bruijn levels [15 points]

In class, we learned about de Bruijn indexes which eliminate names of variables in the λ-calculus and thus dispenses with α-conversion, i.e., renaming in terms. De Bruijn's original article actually contains two different notations for nameless representations of terms: de Bruijn indexes and de Bruijn levels.

Similarly to de Bruijn indexes, de Bruijn levels also use integers, called "levels," to represent variables. The key difference is that de Bruijn levels number binders "from the outside in," whereas de Bruijn indexes number binders "from the inside out." For example, expression $\lambda x.\, (\lambda y.\, \lambda z.\, x\ y\ z)\ (\lambda w.\, w)$ is represented using de Bruijn indexes and de Bruijn levels as follows:

$$\begin{array}{lcll} \lambda x.\, (\lambda y.\, \lambda z.\, x\ y\ z)\ (\lambda w.\, x\ w) & \equiv_{\mathsf{dB_I}} & \lambda.\, (\lambda.\, \lambda.\, 2\ 1\ 0)\ (\lambda.\, 1\ 0) & \text{(using de Bruijn indexes)} \\ \lambda x.\, (\lambda y.\, \lambda z.\, x\ y\ z)\ (\lambda w.\, x\ w) & \equiv_{\mathsf{dB_L}} & \lambda.\, (\lambda.\, \lambda.\, 0\ 1\ 2)\ (\lambda.\, 0\ 1) & \text{(using de Bruijn levels)} \end{array}$$

Recall that a de Bruijn index for a variable specifies the relative position of its corresponding λ-binder. This, in turn, implies that the same variable can be assigned different de Bruijn indexes depending on its position. In contrast, *the* de Bruijn level for a variable specifies the position of its corresponding λ-binder relative to the outermost λ-binder. Hence the same variable is assigned a unique de Bruijn level regardless of its position inside an expression. In the example above, the two instances of the same variable x are assigned the same de Bruijn level 1.

To exploit de Bruijn levels in implementing the operational semantics of the λ-calculus, we introduce a substitution function $\delta_{i,k}(M, N)$ which substitutes N for a variable in M. We define $\delta_{i,k}(M, N)$ so that the following relationship holds; the meaning of subscripts i and k is explained later:

$$\begin{array}{ccc} (\lambda x.\, e)\ e' & \mapsto & [e'/x]e \\ \equiv_{\mathsf{dB_L}} & & \equiv_{\mathsf{dB_L}} \\ (\lambda.\, M)\ N & \mapsto & \delta_{i,k}(M, N) \end{array}$$

That is, applying $\lambda.\, M$ to N, or substituting N for the outermost variable bound in $\lambda.\, M$, results in $\delta_{i,k}(M, N)$. Here is a simple example:

$$\begin{array}{ccccc} (\lambda x.\, x)\ (\lambda y.\, y) & \mapsto & [(\lambda y.\, y)/x]x & = & \lambda y.\, y \\ \equiv_{\mathsf{dB_L}} & & \equiv_{\mathsf{dB_L}} & & \equiv_{\mathsf{dB_L}} \\ (\lambda.\, 0)\ (\lambda.\, 0) & \mapsto & \delta_{0,0}(0, \lambda.\, 0) & = & \lambda.\, 0 \end{array}$$

In $\delta_{i,k}(M, N)$, the subscript i is the de Bruijn level of the variable to be replaced by N. That is, we are to substitute N for de Bruijn level i in M. The subscript k denotes the number of λ-binders encountered so far, *i.e.*, before reaching M. That is, it denotes the number of λ-binders lying between the λ-binder corresponding to the de Bruijn level i and M. Consider the following example in which the redex is underlined:

$$\lambda x.\, \underline{(\lambda y.\, \lambda z.\, y)\ (\lambda w.\, x\ w)} \quad \equiv_{\mathsf{dB_L}} \quad \lambda.\, \underline{(\lambda.\, \lambda.\, 1)\ (\lambda.\, 0\ 1)} \quad \mapsto \quad \lambda.\, \delta_{1,0}(\lambda.\, 1, \lambda.\, 0\ 1)$$

Here the first subscript 1 is the de Bruijn level of variable y, which is to be replaced in this substitution. The second subscript is 0 because there is no intervening λ-binder between λy and $\lambda z.\, y$. We propagate the substitution function δ, its second subscript changes to 1 because the λ-binder λz:

$$\lambda.\, \delta_{1,0}(\lambda.\, 1, \lambda.\, 0\ 1) \quad = \quad \lambda.\, \lambda.\, \delta_{1,1}(1, \lambda.\, 0\ 1)$$

The goal in this problem is to complete the definition of the substitution function $\delta_{i,k}(M, N)$. As with de Bruijn indexes, you will need a shifting function $\tau_i^k(N)$ which shifts by k all de Bruijn levels in N that are greater than or equal to i.

Question 1. **[5 points]** Fill in the blank with appropriate variable names so that the following relation holds:

$$\lambda\_\_\_.\, \lambda\_\_\_.\, (\lambda\_\_\_.\, (\lambda\_\_\_.\, \_\_\_\ \_\_\_)\ \_\_\_)\ (\lambda\_\_\_.\, \_\_\_\ \_\_\_\ \_\_\_) \quad \equiv_{\mathsf{dB_L}} \quad \lambda.\, \lambda.\, (\lambda.\, (\lambda.\, 3\ 2)\ 2)\ (\lambda.\, 0\ 1\ 2)$$

Question 2. **[5 points]** Complete the definition of $\tau_i^k(N)$.

$$\begin{array}{rcll} \tau_i^k(N_1\ N_2) & = & \underline{\qquad\qquad\qquad\qquad\qquad\qquad} & \\ \tau_i^k(\lambda.\, N) & = & \underline{\qquad\qquad\qquad\qquad\qquad} & \\ \tau_i^k(n) & = & \underline{\qquad\qquad\qquad\qquad} & \text{if } n \geq i \\ \tau_i^k(n) & = & \underline{\qquad\qquad\qquad\qquad} & \text{if } n < i \end{array}$$

Question 3. **[5 points]** Use $\tau_i^k(N)$ and complete the definition of $\delta_{i,k}(M, N)$.

$$\delta_{i,k}(M_1\ M_2, N) \quad = \quad \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$\delta_{i,k}(\lambda.\, M, N) \quad = \quad \underline{\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$\delta_{i,k}(m, N) \quad = \quad \underline{\qquad\qquad\qquad\qquad} \qquad \text{if } \ m < i$$

$$\delta_{i,k}(i, N) \quad = \quad \underline{\qquad\qquad\qquad\qquad\qquad}$$

$$\delta_{i,k}(m, N) \quad = \quad \underline{\qquad\qquad\qquad\qquad\qquad} \qquad \text{if } \ m > i$$

5 Simply typed λ-calculus [15 points]

In this problem, we use the following fragment of the simply typed λ-calculus. We do not consider base types. We use A, B, C for metavariables for types, e for expressions, and Γ for typing contexts. We use a typing judgment $\Gamma \vdash e : A$ to mean that under typing context Γ, expression e has type A. We use a reduction judgment $e \mapsto e'$ to mean that expression e reduces to expression e'.

$$\begin{array}{rrcl} \textsf{type} & A & ::= & P \mid A \to A \\ \textsf{base type} & P & & \\ \textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \\ \textsf{value} & v & ::= & \lambda x\!:\!A.\, e \\ \textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsf{Var} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x\!:\!A.\, e : A \to B} \to\!\textsf{I} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B} \to\!\textsf{E}$$

Question 1. [5 points] State two theorems, progress and type preservation, constituting type safety:

(Progress).

(Type preservation).

Question 2. [10 points] Fill in the blank to complete the proof of the substitution lemma where $[e'/x]e$ denotes the capture-avoiding substitution of e' for every occurrence of x in e. We assume that a typing context is an unordered set and that variables in a typing context are all distinct.

Lemma 5.1 (Substitution)**.** *If* $\Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash e' : C$*, then* $\Gamma \vdash [e/x]e' : C$*.*

Proof. By rule induction on the judgment $\Gamma, x : A \vdash e' : C$. In the third case, $FV(e)$ denotes the set of free variables in expression e, and we assume (without loss of generality) that y is a fresh variable such that $y \notin FV(e)$ and $y \neq x$. If $y \in FV(e)$ or $y = x$, we can always choose a different variable by applying an α-conversion to $\lambda y{:}C_1.\, e''$.

Case $\dfrac{y : C \in \Gamma, x : A}{\Gamma, x : A \vdash y : C}\ \mathsf{Var}$ where $e' = y$ and $y : C \in \Gamma$:

$\Gamma \vdash y : C$ from $y : C \in \Gamma$

$[e/x]y = y$ from $x \neq y$

Case $\dfrac{}{\Gamma, x : A \vdash x : A}\ \mathsf{Var}$ where $e' = x$ and $C = A$:

_____ assumption

_____ $[e/x]x = e$

Case $\dfrac{\Gamma, x : A, y : C_1 \vdash e'' : C_2}{\Gamma, x : A \vdash \lambda y{:}C_1.\, e'' : C_1 \to C_2}\ \to\mathsf{I}$ where $e' = \lambda y{:}C_1.\, e''$ and $C = C_1 \to C_2$:

_____ by induction hypothesis

_____ by the rule $\to\mathsf{I}$

$[e/x]\lambda y{:}C_1.\, e'' =$ _____ from $y \notin FV(e)$ and $x \neq y$

_____

Case $\dfrac{\Gamma, x : A \vdash e_1 : B \to C \quad \Gamma, x : A \vdash e_2 : B}{\Gamma, x : A \vdash e_1\ e_2 : C}\ \to\mathsf{E}$ where $e' = e_1\ e_2$:

_____ by induction hypothesis on _____

_____ by induction hypothesis on _____

_____ by the rule $\to\mathsf{E}$

$\Gamma \vdash [e/x](e_1\ e_2) : C$ from _____

□

6 A weird reduction strategy [20 points]

Consider the same fragment of the simply typed λ-calculus in the previous problem. We will develop a weird reduction strategy specified as follows:

- Given an application $e_1\ e_2$, we first reduce e_2.
- After reducing e_2 to a value, we reduce e_1.

- When e_1 reduces to a λ-abstraction, we apply the β-reduction.

We write $[e'/x]e$ for the capture-avoiding substitution of e' for every occurrence of x in e.

Question 1. [5 points] Give the reduction rules for the reduction judgment $e \mapsto e'$ under the weird reduction strategy. You need three reduction rules.

$$\frac{\quad}{\quad \mapsto \quad} \qquad \frac{\quad}{\quad \mapsto \quad} \qquad \frac{\quad}{\quad \mapsto \quad}$$

Question 2. [5 points] Give the definition of evaluation contexts corresponding to the weird reduction strategy:

evaluation context $\quad \kappa \quad ::=$ ____________________________________

Question 3. [5 points] We wish to develop an abstract machine C for the weird reduction strategy. The abstract machine C uses two states:

state $\quad s \quad ::= \quad \sigma \blacktriangleright e \mid \sigma \blacktriangleleft v$

- $\sigma \blacktriangleright e$ means that the machine is currently reducing $\sigma[\![e]\!]$, but has yet to analyze e.
- $\sigma \blacktriangleleft v$ means that the machine is currently reducing $\sigma[\![v]\!]$ and has already analyzed v.

Give the definitions of frames and stacks:

frame $\quad \phi \quad ::=$ ____________________________________

stack $\quad \sigma \quad ::=$ ____________________________________

Question 4. [5 points] Give the rules for the reduction judgment $s \mapsto_{\mathsf{C}} s'$ for the abstract machine C.