Name: Hemos ID:

CSE-321 Programming Languages 2014
Final

	Problem 1	Problem 2	Problem 3	Problem 4	Problem 5	Problem 6	Total
Score							
Max	15	12	14	14	30	15	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.
- You have three hours for this exam.

1 Inductive definitions of matched parentheses [15 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. Here are a couple of examples:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \rhd \epsilon}\ \mathit{Peps} \quad 1 > 0}{1 \rhd)}\ \mathit{Pright} \quad 2 > 0}{2 \rhd))}\ \mathit{Pright}}{1 \rhd ())}\ \mathit{Pleft}}{0 \rhd (())}\ \mathit{Pleft}$$

(*the rule Pright is not applicable because* $0 \not> 0$)

$$\dfrac{\dfrac{\dfrac{0 \rhd)(}{1 \rhd))(}\ \mathit{Pright}}{0 \rhd ())(}\ \mathit{Pleft}$$

We wish to prove Theorem 1.2 which states that a string s satisfying $0 \rhd s$ indeed belongs to the syntactic category mparen. We use a lemma 1.1 in the proof.

Lemma 1.1. *For every natural number* $k \geq 0$, *if* $\underbrace{((\cdots(}_{k}\, s$ mparen, *then* $\underbrace{((\cdots(}_{k}()s$ mparen.

Theorem 1.2. *If* $0 \rhd s$, *then* s mparen.

Prove Theorem 1.2. You may use Lemma 1.1 without proving it. In your proof, place *conclusion* in the left and *justification* in the right as is conventional in the course notes. *If we fail to understand your proof, you get no credit, so try to make your proof as readable as possible.* You may write either $\underbrace{((\cdots(}_{k}$ or $(^k$ for a sequence of k ('s.

(Answer sheet for Problem 1)

2 Abstract machine N [12 points]

In this problem, we design an abstract machine N based on the call-by-need reduction strategy which you implemented in Assignment 6. The call-by-need reduction strategy is a variant of the call-by-name strategy: it reduces a function application without evaluating the argument, but is also designed so that it never evaluates the same argument more than once. To this end, it delays the evaluation of the argument until the result is needed. Once the argument is fully evaluated, it stores the result in the heap so that when it needs the argument again, it does not need to repeat the same evaluation.

The heap stores two different kinds of objects: *delayed expressions* and *computed values*. When the machine attempts to evaluates a function application, it first allocates a new delayed expression for the argument in the heap and then proceeds to reducing the function application. When the machine later evaluates the variable bound to the argument for the first time, it retrieves the actual argument from the delayed expression to evaluates it. Then it replaces the delayed expression with a computed value in the heap so that all subsequent references to the same variable can directly use the result without repeating the same evaluation.

The abstract machine N uses the following definitions:

type	A	::=	$P \mid A \to A$
expression	e	::=	$x \mid \lambda x{:}A.\, e \mid e\ e$
value	v	::=	_____
stored value	sv	::=	_____
heap	h	::=	$\cdot \mid h, l \hookrightarrow sv$
environment	η	::=	$\cdot \mid \eta, x \hookrightarrow l$
frame	ϕ	::=	_____
stack	σ	::=	$\square \mid \sigma; \phi$
state	s	::=	$h \parallel \sigma \blacktriangleright e \mathbin{@} \eta \ \mid\ h \parallel \sigma \blacktriangleleft v$

A value v contains the result of evaluating an expression. A stored value sv is an object to be stored in the heap h, and thus is either a delayed expression or a computed value.

In the definition of state s:

- $h \parallel \sigma \blacktriangleright e \mathbin{@} \eta$ means that the machine with heap h and stack σ is currently analyzing e under environment η.
- $h \parallel \sigma \blacktriangleleft v$ means that the machine with heap h and stack σ is currently returning v.

The reduction judgment for the abstract machine N is as follows:

$$s \mapsto_{\mathsf{N}} s' \qquad \Leftrightarrow \qquad \textit{the machine makes a transition from state } s \textit{ to another state } s'$$

To define the operational semantics, we use a couple of auxiliary functions. First we define the function $dom(h)$ which returns the set of all locations in a given heap h:

$$\begin{array}{rcl} dom(\cdot) & = & \varnothing \\ dom(h, l \hookrightarrow sv) & = & dom(h) \cup \{l\} \end{array}$$

We also use $[l \hookrightarrow sv]h$ for updating the contents of l in h with sv:

$$[l \hookrightarrow sv'](h, l \hookrightarrow sv) = h, l \hookrightarrow sv'$$

Complete the definitions of value v, stored value sv, and frame ϕ. Then define transition rules for the abstract machine N. You may introduce as many transition rules as you need. Explain your definitions and transition rules as necessary.

(Definitions)

value	v	::=	_____
stored value	sv	::=	_____
frame	ϕ	::=	_____

(Transition rules)

3 Subtypes and recursive types [14 points]

Question 1. [3 points] The rule of *subsumption* is a typing rule which enables us to change the type of an expression to its subtype. Complete the rule of subsumption:

Question 2. [3 points] Complete the subtyping rule for product types and function types.

$$\frac{}{A \times B \leq A' \times B'}\ \mathit{Prod}_{\leq}$$

$$\frac{}{A \to B \leq A' \to B'}\ \mathit{Fun}_{\leq}$$

Question 3. [3 points] Consider the following simply typed λ-calculus extended with recursive types.

$$\begin{array}{rrcl} \mathsf{type} & A & ::= & \mathsf{unit} \mid A \to A \mid A + A \mid \alpha \mid \mu\alpha.A \\ \mathsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid () \mid \mathsf{inl}_A\ e \mid \mathsf{inr}_A\ e \mid \\ & & & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.\, e \mid \mathsf{inr}\ x.\, e \mid \mathsf{fold}_C\ e \mid \mathsf{unfold}_C\ e \\ \mathsf{typing\ context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \mathsf{type} \end{array}$$

Given a recursive type $C = \mu\alpha.A$, $\mathsf{fold}_C\ e$ and $\mathsf{unfold}_C\ e$ convert $[C/\alpha]A$ to C and vice versa, respectively. Complete typing rules for $\mathsf{fold}_C\ e$ and $\mathsf{unfold}_C\ e$:

$$\frac{C = \mu\alpha.A \qquad \Gamma \vdash C\ \mathsf{type}}{\Gamma \vdash \mathsf{fold}_C\ e :}\ \mathsf{Fold}$$

$$\frac{C = \mu\alpha.A}{\Gamma \vdash \mathsf{unfold}_C\ e :}\ \mathsf{Unfold}$$

Question 4. [5 points] Translate constructs for a recursive datatype for natural numbers into the above simply typed λ-calculus.

$$\begin{array}{rcl} \texttt{datatype nat} & = & \texttt{Zero} \mid \texttt{Succ of nat} \\ \texttt{nat} & = & \underline{\qquad\qquad} \\ \texttt{Zero} & = & \underline{\qquad\qquad} \\ \texttt{Succ}\ e & = & \underline{\qquad\qquad} \\ \texttt{case}\ e\ \texttt{of Zero} \Rightarrow e_1 \mid \texttt{Succ}\ x \Rightarrow e_2 & = & \underline{\qquad\qquad} \end{array}$$

4 System F [14 points]

Consider the following definitions for System F:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & A \to A \mid \alpha \mid \forall \alpha . A \\ \textsf{expression} & e & ::= & x \mid \lambda x : A . \, e \mid e \; e \mid \Lambda \alpha . \, e \mid e \, [\![A]\!] \\ \textsf{value} & v & ::= & \lambda x : A . \, e \mid \Lambda \alpha . \, e \\ \textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha \; \textsf{type} \end{array}$$

Note that a typing context Γ is an *ordered* set of type bindings and type declarations. We use three judgments: a reduction judgment, a type judgment, and a typing judgment.

$$\begin{array}{rcl} e \mapsto e' & \Leftrightarrow & e \text{ reduces to } e' \\ \Gamma \vdash A \; \textsf{type} & \Leftrightarrow & A \text{ is a valid type with respect to typing context } \Gamma \\ \Gamma \vdash e : A & \Leftrightarrow & e \text{ has type } A \text{ under typing context } \Gamma \end{array}$$

Question 1. **[4 points]** Write two reduction rules for type applications $e \, [\![A]\!]$:

Question 2. **[4 points]** Write the typing rules for type abstractions $\Lambda \alpha . \, e$ and type applications $e \, [\![A]\!]$:

Question 3. **[6 points]** The proof of type safety of System F needs three substitution lemmas because there are three kinds of substitutions in System F: type substitution into types, type substitution into expressions, and expression substitution. State the three substitution lemmas for the proof of type safety of System F.

1. For substituting types for type variables in types:

2. For substituting types for type variables in expressions:

3. For substituting expressions for variables in expressions:

5 Type reconstruction [30 points]

Consider the implicit let-polymorphic type system given in the course notes.

$$
\begin{array}{rrcl}
\textsf{monotype} & A & ::= & A \to A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall \alpha . U \\
\textsf{expression} & e & ::= & x \mid \lambda x.\, e \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : U \\
\textsf{type substitution} & S & ::= & \mathrm{id} \mid \{A/\alpha\} \mid S \circ S \\
\textsf{type equations} & E & ::= & \cdot \mid E, A = A
\end{array}
$$

- We use a typing judgment $\Gamma \triangleright e : U$ to express that untyped expression e is typable with a polytype U under typing context Γ.
- $S \cdot U$ and $S \cdot \Gamma$ denote applications of S to U and Γ, respectively.
- $\mathit{ftv}(\Gamma)$ denotes the set of free type variables in Γ. $\mathit{ftv}(U)$ denotes the set of free type variables in U.
- An auxiliary function $\textsf{Gen}_\Gamma(A)$ generalizes monotype A to a polytype after taking into account free type variables in typing context Γ:

$$\textsf{Gen}_\Gamma(A) \ = \ \forall \alpha_1 . \forall \alpha_2 . \cdots \forall \alpha_n . A \text{ where } \alpha_i \notin \mathit{ftv}(\Gamma) \text{ and } \alpha_i \in \mathit{ftv}(A) \text{ for } i = 1, \cdots, n.$$

- We write $\Gamma + x : U$ for $\Gamma - \{x : U'\}, x : U$ if $x : U' \in \Gamma$, and for $\Gamma, x : U$ if Γ contains no type binding for variable x.
- The function $\textsf{Unify}$ has a property that if $\textsf{Unify}(A_1 = A_1', \cdots, A_n = A_n') = S$, then $S \cdot A_i = S \cdot A_i'$ for $i = 1, \cdots, n$.

Question 1. **[5 points]** Write all the typing rules for the typing judgment $\Gamma \triangleright e : U$. In the rule for specialization, you may use a type judgment $\Gamma \vdash A\ \textsf{type}$ to mean that A is a valid type with respect to typing context Γ.

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

Question 2. **[3 points]** Fill in the blank:

$$\begin{aligned}
\mathsf{Gen}_{\cdot}(\alpha \to \alpha) &= \forall \alpha.\alpha \to \alpha \\
\mathsf{Gen}_{x:\alpha}(\alpha \to \alpha) &= \underline{\qquad\qquad\qquad\qquad} \\
\mathsf{Gen}_{x:\alpha}(\alpha \to \beta) &= \underline{\qquad\qquad\qquad\qquad} \\
\mathsf{Gen}_{x:\alpha, y:\beta}(\alpha \to \beta) &= \underline{\qquad\qquad\qquad\qquad}
\end{aligned}$$

Question 3. **[6 points]** Complete the unification algorithm Unify:

$$\begin{aligned}
\mathsf{Unify}(\cdot) &= \mathrm{id} \\
\mathsf{Unify}(E, \alpha = A) = \mathsf{Unify}(E, A = \alpha) &= \text{if } \alpha = A \text{ then } \underline{\qquad\qquad} \\
&\qquad \text{else if } \underline{\qquad\qquad\qquad} \text{ then } \mathit{fail} \\
&\qquad \text{else } \underline{\qquad\qquad\qquad\qquad} \\
\mathsf{Unify}(E, A_1 \to A_2 = B_1 \to B_2) &= \underline{\qquad\qquad\qquad\qquad}
\end{aligned}$$

Question 4. **[12 points]** Complete the algorithm $\mathcal{W}$. Its soundness means that if $\mathcal{W}(\Gamma, e) = (S, A)$, then $S \cdot \Gamma \triangleright e : A$.

$$\mathcal{W}(\Gamma, x) = (\mathrm{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A) \qquad \text{where } x : \forall\vec{\alpha}.A \in \Gamma \text{ and fresh } \vec{\beta}$$

$$\mathcal{W}(\Gamma, \lambda x.\, e) = \text{let } (S, A) = \mathcal{W}(\Gamma + x : \alpha, e) \text{ in} \qquad (\text{fresh } \alpha)$$

$$\_\_\_\_\_$$

$$\mathcal{W}(\Gamma, e_1\ e_2) = \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in}$$

$$\text{let } \_\_\_\_\_ \text{ in}$$

$$\text{let } S_3 = \_\_\_\_\_ \text{ in} \qquad (\text{fresh } \alpha)$$

$$(\_\_\_\_\_, \_\_\_\_\_)$$

$$\mathcal{W}(\Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in}$$

$$\text{let } (S_2, A_2) = \_\_\_\_\_ \text{ in}$$

$$(\_\_\_\_\_, \_\_\_\_\_)$$

Question 5. **[4 points]** Now we add an untyped fixed point construct $\mathsf{fix}\ x.\, e$. The typing rule for $\mathsf{fix}\ x.\, e$ is as follows:

$$\frac{\Gamma, x : A \triangleright e : A}{\Gamma \triangleright \mathsf{fix}\ x.\, e : A}\ \mathsf{Fix}$$

Complete the case for $\mathsf{fix}\ x.\, e$ in the algorithm $\mathcal{W}$:

$$\mathcal{W}(\Gamma, \mathsf{fix}\ x.\, e) = \text{let } (S_1, A_1) = \_\_\_\_\_ \text{ in} \qquad (\text{fresh } \alpha)$$

$$\text{let } S_2 = \_\_\_\_\_ \text{ in}$$

$$(\_\_\_\_\_, \_\_\_\_\_)$$

6 Soundness of the algorithm $\mathcal{W}$ [15 points]

Prove the soundness of the algorithm $\mathcal{W}$. You may use the following three lemmas without proofs:

Lemma 6.1. *If* $\Gamma \rhd e : A$, *then* $S \cdot \Gamma \rhd e : S \cdot A$ *for any type substitution* S.

Lemma 6.2. *If* $\mathsf{Unify}(A_1 = A'_1, \cdots, A_n = A'_n) = S$, *then* $S \cdot A_i = S \cdot A'_i$ *for* $i = 1, \cdots, n$.

Lemma 6.3. *If* $\Gamma \rhd e : A$, *then* $\Gamma \rhd e : \mathsf{Gen}_\Gamma(A)$.

Complete the proof of Theorem 6.4.

Theorem 6.4 (Soundness of $\mathcal{W}$)**.** *If* $\mathcal{W}(\Gamma, e) = (S, A)$, *then* $S \cdot \Gamma \rhd e : A$.

Proof. By structural induction on e. We consider three cases shown below. We give a complete proof of the first case and a partial proof of the second case.

Case $e = \lambda x.\, e'$:

There exist S_1, A_1, and a fresh type variable α_1 such that:
$\mathcal{W}(\Gamma + x : \alpha_1, e') = (S_1, A_1)$ by the definition of $\mathcal{W}$

$S_1 \cdot \Gamma + x : S_1 \cdot \alpha_1 \rhd e' : A_1$ by induction hypothesis on e'

$$\left.\begin{array}{l} \mathcal{W}(\Gamma, \lambda x.\, e') = (S_1, (S_1 \cdot \alpha_1) \to A_1) \\ S = S_1 \\ A = (S_1 \cdot \alpha_1) \to A_1 \end{array}\right\}$$ by the definition of $\mathcal{W}$

$S_1 \cdot \Gamma \rhd \lambda x.\, e' : (S_1 \cdot \alpha_1) \to A_1$ by the rule $\to\mathsf{I}$ with $S_1 \cdot \Gamma + x : S_1 \cdot \alpha_1 \rhd e' : A_1$

$S \cdot \Gamma \rhd \lambda x.\, e' : A$ from $S = S_1$, $A = (S_1 \cdot \alpha_1) \to A_1$

Case $e = e_1\ e_2$:

There exist S_1 and A_1 such that:
$\mathcal{W}(\Gamma, e_1) = (S_1, A_1)$ by the definition of $\mathcal{W}$

$S_1 \cdot \Gamma \rhd e_1 : A_1$ by induction hypothesis on e_1

There exist S_2 and A_2 such that:
$\mathcal{W}(S_1 \cdot \Gamma, e_2) = (S_2, A_2)$ by the definition of $\mathcal{W}$

$S_2 \cdot S_1 \cdot \Gamma \rhd e_2 : A_2$ by induction hypothesis on e_2

Case $e \;=\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: