Q1	Q2	Q3	Q4	Q5	TOTAL
50	50	50	45	30	225

bonus

Definition of Programming Languages

CPSC 311 2016W1

University of British Columbia

Practice Final Examination—Episode One **plus Q5**

Joshua Dunfield

Student Name:

Student Number:

Signature:

INSTRUCTIONS

- This is a CLOSED BOOK / CLOSED NOTES examination.
- Write all answers ON THE EXAMINATION PAPER.

  Try to write your answers in the blanks or boxes given. If you can't, try to write them elsewhere on the same page, or on one of the worksheet pages, and **LABEL THE ANSWER**. We can't give credit for answers we can't find.
- Blanks are suggestions. You may not need to fill in every blank.

This is an **incomplete** practice exam; the actual final exam will have an additional question on subtyping. We plan to release an updated version of this practice exam that includes a question on subtyping.

Question 1 [50 points]: "If it's 'dynamic', it *must* be better."

The following rules define an environment-based semantics for lexically-scoped functions, Lam, and dynamically-scoped functions, Ds-lam.

$\boxed{env \vdash e \Downarrow v}$ Under environment env, expression e evaluates to value v

$$\frac{}{env \vdash (\textsf{Num}\ n) \Downarrow (\textsf{Num}\ n)}\ \text{Env-num} \qquad \frac{env \vdash e1 \Downarrow (\textsf{Num}\ n1) \qquad env \vdash e2 \Downarrow (\textsf{Num}\ n2)}{env \vdash (\textsf{Add}\ e1\ e2) \Downarrow (\textsf{Num}\ n1{+}n2)}\ \text{Env-add}$$

$$\frac{env \vdash e1 \Downarrow v1 \qquad x{=}v1, env \vdash e2 \Downarrow v2}{env \vdash (\textsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Env-let}$$

$$\frac{lookup(env, x) = e}{env \vdash (\textsf{Id}\ x) \Downarrow e}\ \text{Env-id} \qquad \frac{lookup(env, x)\ \text{undefined}}{env \vdash (\textsf{Id}\ x)\ \textsf{unknown-id-error}}\ \text{Env-unknown-id}$$

$$\frac{}{env \vdash (\textsf{Lam}\ x\ e1) \Downarrow \big(\textsf{Clo}\ env\ (\textsf{Lam}\ x\ e1)\big)}\ \text{Env-lam} \qquad \frac{env_{\text{old}} \vdash e \Downarrow v}{env \vdash (\textsf{Clo}\ env_{\text{old}}\ e) \Downarrow v}\ \text{Env-clo}$$

$$\frac{env \vdash e1 \Downarrow (\textsf{Clo}\ env_{\text{old}}\ (\textsf{Lam}\ x\ eB)) \qquad env \vdash e2 \Downarrow v2 \qquad x{=}v2, env_{\text{old}} \vdash eB \Downarrow v}{env \vdash (\textsf{App}\ e1\ e2) \Downarrow v}\ \text{Env-app}$$

$$\frac{}{env \vdash (\textsf{Ds-lam}\ x\ e1) \Downarrow (\textsf{Ds-lam}\ x\ e1)}\text{Env-ds-lam} \quad \frac{env \vdash e1 \Downarrow (\textsf{Ds-lam}\ x\ eB) \quad env \vdash e2 \Downarrow v2 \quad x{=}v2, env \vdash eB \Downarrow v}{env \vdash (\textsf{App}\ e1\ e2) \Downarrow v}\text{Env-ds-app}$$

Assume that $lookup(env, x)$ returns the **leftmost** binding of x. For example:

$$lookup\Big(\big(x{=}(\textsf{Num}\ 2), x{=}(\textsf{Num}\ 1), \emptyset\big), x\Big) \;=\; (\textsf{Num}\ 2)$$

Consider the following expression, shown in concrete syntax (left) and in abstract syntax (right).

```
{Let y 100
     {Let f {Let y 10
                 {Lam x {+ x y}}}
            {Let y 2
                 {App f y}}}}
```

$$\Big(\textsf{Let}\ y\ (\textsf{Num}\ 100)\ \Big(\textsf{Let}\ f\ \big(\textsf{Let}\ y\ (\textsf{Num}\ 10)\ (\textsf{Lam}\ x\ (\textsf{Add}\ (\textsf{Id}\ x)\ (\textsf{Id}\ y)))\big)\ \big(\textsf{Let}\ y\ (\textsf{Num}\ 2)\ (\textsf{App}\ (\textsf{Id}\ f)\ (\textsf{Id}\ y))\big)\Big)\Big)$$

Q1a [10 points] Complete the **abstract syntax tree** for the above expression. $\Longrightarrow$

```
   Let
  / |  \
 y Num
   100
```

Question 1 [50 points]: "If it's 'dynamic', it *must* be better." (cont.)

Q1b [10 points] If we evaluate the above expression in the empty environment, what value do we get?

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Q1c [10 points] While evaluating the above expression, we will evaluate the body of the Lam. When we evaluate that expression, (Add (Id x) (Id y)), what is the complete environment?

_ _

Q1d [10 points] **Warning:** Dynamic scope ahead!

If we evaluate the expression that is **almost** the same, but has Ds-lam in place of Lam, as shown **below**, what value do we get? _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

```
{Let y 100
 {Let f {Let y 10
              {Ds-lam x {+ x y}}}
       {Let y 2
            {App f y}}}}
```

```
(Let y (Num 100)
  (Let f (Let y (Num 10)
                (Ds-lam x (Add (Id x) (Id y))))
         (Let y (Num 2)
                (App (Id f) (Id y)))))
```

Q1e [10 points] (No more dynamic scope. Yay!)

This will be a question about substitution.
It should be roughly similar to Q1 on the midterm, **but will probably ask you to show your work by writing the intermediate steps of applying the substitution**.

Worksheet (i)

Worksheet (ii)

Question 2 [50 points]: Little Perennials II

The *expression strategy*, *value strategy*, and *lazy evaluation* are different ways of evaluating a function application (App $e1$ $e2$). All strategies evaluate $e1$ to (Lam x eB), but they differ in how they handle $e2$:

- The expression strategy evaluates eB with x bound to a "thunk" containing $e2$. The thunk is a closure (Clo env $e2$), which saves the current environment env, to make sure we *don't* use dynamic scoping.
- The value strategy evaluates $e2$ to a value $v2$, then evaluates eB with x bound to that value.
- Lazy evaluation creates a *lazy thunk*, $\ell \triangleright$(Lazy-thk env $e2$), in the store, and evaluates eB with x bound to (Lazy-ptr ℓ). If (Id x) is evaluated, we evaluate $e2$ to a value $v2$, and replace $\ell \triangleright$(Lazy-thk env $e2$) with $\ell \triangleright v2$ (rule SEnv-lazy-ptr). If (Id x) is evaluated again, rule SEnv-lazy-ptr-done looks up the value $v2$, without evaluating $e2$ again.

If you need to, you can refer to the following evaluation rules:

$\boxed{env; S \vdash e \Downarrow v; S'}$ Under environment env and store S, expression e evaluates to v with updated store S'

$$\frac{}{env; S \vdash (\mathsf{Lam}\ x\ e1) \Downarrow \big(\mathsf{Clo}\ env\ (\mathsf{Lam}\ x\ e1)\big); S}\ \text{SEnv-lam} \qquad \frac{env_{old}; S1 \vdash e \Downarrow v; S2}{env; S1 \vdash (\mathsf{Clo}\ env_{old}\ e) \Downarrow v; S2}\ \text{SEnv-clo}$$

$$\frac{env; S \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ x\ eB)); S1 \qquad env; S1 \vdash e2 \Downarrow v2; S2 \qquad x{=}v2, env_{old}; S2 \vdash eB \Downarrow v; S'}{env; S \vdash (\mathsf{App}\ e1\ e2) \Downarrow v; S'}\ \text{SEnv-app-}\textbf{value}$$

$$\frac{env; S \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ x\ eB)); S1 \qquad x{=}(\mathsf{Clo}\ env\ e2), env_{old}; S1 \vdash eB \Downarrow v; S2}{env; S \vdash (\mathsf{App}\ e1\ e2) \Downarrow v; S2}\ \text{SEnv-app-}\textbf{expr}$$

$$\frac{env; S \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ x\ eB)); S1 \qquad x{=}(\mathsf{Lazy\text{-}ptr}\ \ell), env_{old}; \ell \triangleright (\mathsf{Lazy\text{-}thk}\ env\ e2),\ S1 \vdash eB \Downarrow v; S2}{env; S \vdash (\mathsf{App}\ e1\ e2) \Downarrow v; S2}\ \text{SEnv-app-}\textbf{lazy}$$

$$\frac{\textit{lookup-loc}(S, \ell) = (\mathsf{Lazy\text{-}thk}\ env_{arg}\ e2) \qquad env_{arg}; S \vdash e2 \Downarrow v; S1 \qquad \textit{update-loc}(S1, \ell, v) = S2}{env; S \vdash (\mathsf{Lazy\text{-}ptr}\ \ell) \Downarrow v; S2}\ \text{SEnv-lazy-ptr}$$

$$\frac{\textit{lookup-loc}(S, \ell) = v2 \qquad v2 \neq (\mathsf{Lazy\text{-}thk}\ \cdots\ \cdots)}{env; S \vdash (\mathsf{Lazy\text{-}ptr}\ \ell) \Downarrow v2; S}\ \text{SEnv-lazy-ptr-done}$$

$$\frac{\textit{lookup}(env, x) = e \qquad env; S \vdash e \Downarrow v; S'}{env; S \vdash (\mathsf{Id}\ x) \Downarrow v; S'}\ \text{SEnv-id}$$

$$\frac{env; S \vdash e1 \Downarrow (\mathsf{Num}\ n1); S1 \qquad env; S1 \vdash e2 \Downarrow (\mathsf{Num}\ n2); S'}{env; S \vdash (\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 + n2); S'}\ \text{SEnv-add}$$

$$\frac{env; S \vdash e1 \Downarrow (\mathsf{Num}\ n1); S1 \qquad env; S1 \vdash e2 \Downarrow (\mathsf{Num}\ n2); S'}{env; S \vdash (\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 - n2); S'}\ \text{SEnv-sub}$$

Question 2 [50 points]: Little Perennials II, continued

Consider the following expression:

```
{App {App {Lam x {Lam y {+ y y}}}
          {- 7 1}}
     {- 10 1}}
```

$$\Big(\mathsf{App}\ \Big(\mathsf{App}\ \big(\mathsf{Lam}\ x\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\big)\ \big(\mathsf{Sub}\ (\mathsf{Num}\ 7)\ (\mathsf{Num}\ 1)\big)\Big)\ \big(\mathsf{Sub}\ (\mathsf{Num}\ 10)\ (\mathsf{Num}\ 1)\big)\Big)$$

Q2a [15 points] If we implement the **SEnv-app-value** rule (and *not* the SEnv-app-expr and SEnv-app-lazy rules) and evaluate the above expression, we will perform

____ addition(s), and ____ subtraction(s).

Q2b [10 points] Now we switch from the value strategy to the expression strategy.
Complete the derivation tree for the second premise of SEnv-app-expr.

$$\dfrac{\dfrac{}{\emptyset;\emptyset \vdash (\mathsf{Lam}\ x\ eBody) \Downarrow (\mathsf{Clo}\ \emptyset\ (\mathsf{Lam}\ x\ eBody));\emptyset}\text{SEnv-lam} \qquad \boxed{\dfrac{}{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}}}{\emptyset;\emptyset \vdash \Big(\mathsf{App}\ \big(\mathsf{Lam}\ x\ \underbrace{(\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))}_{eBody}\big)\ \underbrace{(\mathsf{Sub}\ (\mathsf{Num}\ 7)\ (\mathsf{Num}\ 1))}_{eSub}\Big) \Downarrow \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}\text{SEnv-app-}\textbf{expr}$$

Q2c [10 points] If we implement the **SEnv-app-expr** rule **instead** of SEnv-app-value, and evaluate the expression at the top of the page, we will perform

____ addition(s), and ____ subtraction(s).

Q2d [15 points] If we implement the **SEnv-app-lazy** rule **instead** of SEnv-app-value, and evaluate the expression at the top of the page, we will perform

____ addition(s), and ____ subtraction(s).

Question 3 [50 points]: Big Log

The small-step semantic interpreters we have seen so far don't include side effects. Useful side effects include state (Ref, Deref, Setref) and input/output.

An interpreter can be extended with input/output by introducing a `Print` construct, and we can model the effect of printing by appending to an output buffer B:

$\boxed{B; e \longrightarrow B'; e'}$ With starting buffer B, expression e steps to expression e' and an updated buffer B'

Reduction rules:

$$\frac{}{B; \big(\mathsf{Add}\ (\mathsf{Num}\ n1)\ (\mathsf{Num}\ n2)\big) \longrightarrow B; (\mathsf{Num}\ n1{+}n2)}\ \text{Step-add}$$

$$\frac{}{B; (\mathsf{Let}\ x\ v1\ e2) \longrightarrow B; \big[v1/x\big]e2}\ \text{Step-let}$$

$$\frac{B2 = \mathit{append}(B1, v)}{B1; (\mathsf{Print}\ v) \longrightarrow B2; v}\ \text{Step-print}$$

Context rule:

$$\frac{B; e \longrightarrow B'; e'}{B; \mathcal{C}[e] \longrightarrow B'; \mathcal{C}[e']}\ \text{Step-context}$$

Rule Step-context uses the following evaluation contexts:

$$\begin{aligned} \mathcal{C} \ ::=\ & [\,] \\ \mid\ & (\mathsf{Add}\ \mathcal{C}\ e) \\ \mid\ & (\mathsf{Add}\ v\ \mathcal{C}) \\ \mid\ & (\mathsf{Let}\ x\ \mathcal{C}\ e) \\ \mid\ & (\mathsf{Print}\ \mathcal{C}) \end{aligned}$$

This question uses the following **define-type**s, and functions with the following signatures:

```
(define-type Config   ; Type of result of 'reduce' and 'step':
  [config (B Buffer?) (e E?)])   ; a buffer and an expression

(define-type Buffer
  [buffer/empty]
  [buffer/append (head Buffer?) (tail E?)])

; append-buffer : Buffer E → Buffer
;
; (append-buffer B1 v) = append(B1, v)

; value? : E → boolean
;
; Returns #true iff e is a Num.
```

Go to the next page.

Question 3 [50 points]: Big Log, continued

Q3a [15 points] In the function `reduce` (below), implement the rule Step-print.

```
; reduce : Buffer E -> (or Config false)
; Given a buffer B and expression e, return (config B2 e2) where B; e ⟶ B2; e2
; using a reduction rule, or #false if no reduction rule can be applied.
(define (reduce B e)
  (type-case E e
    [Add (e1 e2)
         (if (and (value? e1) (Num? e1)
                  (value? e2) (Num? e2))
             (config B
                     (Num (+ (Num-n e1) (Num-n e2))))
             #false)]
    ; ...
    [Print (e1)

       ----------------------------------------------------

       ----------------------------------------------------

       ----------------------------------------------------

       ----------------------------------------------------
    ]))
```

Q3b [15 points] In the function `step` (below), implement the evaluation context (Print $\mathcal{C}$).

```
; step : Buffer E -> (or Config false)
; Given a buffer B and expression e, return (config B2 e2) where B; e ⟶ B2; e2
; using rule Step-context, or #false if no derivation of B; e ⟶ B2; e2 exists.
(define (step B e)
  (or (reduce B e)
      (type-case E e
        [Add (e1 e2)
           (if (step B e1)
               (type-case Config (step B e1)   ; C ::= (Add C e2)
                 [config (B2 s1)
                         (config B2 (Add s1 e2))])
               (if (and (value? e1) (step B e2))
                   (type-case Config (step B e2)   ; C ::= (Add v C)
                     [config (B2 s2)
                             (if s2
                                 (config B2 (Add e1 s2))
                                 #false)])
                   #false)
               )])]
        ; ...
        [Print (e1)

          -------------------------------------------------------

          -------------------------------------------------------

          -------------------------------------------------------

          -------------------------------------------------------

          -------------------------------------------------------  ])))
```

Question 3 [50 points]: Big Log, continued

Q3c [20 points] We can model concurrency in our language by adding angelic nondeterminism:

$$\frac{}{\mathsf{B}; (\mathsf{Par}\ v1\ e2) \longrightarrow \mathsf{B}; v1}\ \text{Step-par-left} \qquad \mathcal{C} ::= \cdots$$

$$\frac{}{\mathsf{B}; (\mathsf{Par}\ e1\ v2) \longrightarrow \mathsf{B}; v2}\ \text{Step-par-right} \qquad \mid (\mathsf{Par}\ \mathcal{C}\ e) \quad \mid (\mathsf{Par}\ e\ \mathcal{C})$$

Let $\longrightarrow^*$ be the reflexive-transitive closure of $\longrightarrow$.

That is, $e \longrightarrow^* e'$ if either $e' = e$, or $e \longrightarrow e2$ and $e2 \longrightarrow^* e'$.

Suppose our program is this expression e:

$$e \ = \ \Big(\mathsf{Print}\ \big(\mathsf{Par}\ (\mathsf{Let}\ r2\ (\mathsf{Print}\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 22))\ (\mathsf{Let}\ r3\ (\mathsf{Print}\ (\mathsf{Num}\ 3))\ (\mathsf{Num}\ 33))\big)\Big)$$

The result of repeatedly stepping e is not always the same; both the buffer and the resulting value can vary.

For example:

$$\langle\ \rangle; e \longrightarrow^* \langle(\mathsf{Num}\ 2), (\mathsf{Num}\ 22)\rangle; (\mathsf{Num}\ 22)$$

by the intermediate steps

$$\begin{aligned}
\langle\ \rangle; e &\longrightarrow \langle(\mathsf{Num}\ 2)\rangle; \Big(\mathsf{Print}\ \big(\mathsf{Par}\ (\mathsf{Let}\ r2\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 22))\ (\mathsf{Let}\ r3\ (\mathsf{Print}\ (\mathsf{Num}\ 3))\ (\mathsf{Num}\ 33))\big)\Big) \\
&\longrightarrow \langle(\mathsf{Num}\ 2)\rangle; \Big(\mathsf{Print}\ \big(\mathsf{Par}\ (\mathsf{Num}\ 22)\ (\mathsf{Let}\ r3\ (\mathsf{Print}\ (\mathsf{Num}\ 3))\ (\mathsf{Num}\ 33))\big)\Big) \\
&\longrightarrow \langle(\mathsf{Num}\ 2)\rangle; \big(\mathsf{Print}\ (\mathsf{Num}\ 22)\big) \\
&\longrightarrow \langle(\mathsf{Num}\ 2), (\mathsf{Num}\ 22)\rangle; (\mathsf{Num}\ 22)
\end{aligned}$$

Fill in the intermediate computation steps:

$$\langle\ \rangle; e \longrightarrow \langle(\mathsf{Num}\ 3)\rangle; \Big(\mathsf{Print}\ \big(\mathsf{Par}\ (\mathsf{Let}\ r2\ (\mathsf{Print}\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 22))\ (\mathsf{Let}\ r3\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 33))\big)\Big)$$

$\longrightarrow$ ______________________________

$\longrightarrow$ ______________________________

$\longrightarrow$ ______________________________

$\longrightarrow$ ______________________________

Note: Different solutions are possible!

Worksheet (iii)

Question 4 [45 points]: Power of Two

This question is about *intersection types*, which are a little like polymorphic types: they describe expressions that have *more than one type*. (But you can answer this question without remembering anything about polymorphic types!)

In *bidirectional typing*, the judgment form $\Gamma \vdash e : A$ is replaced by two judgments:

$\Gamma \vdash e \Rightarrow A$ read "under assumptions in Γ, the expression e synthesizes type A"

$\Gamma \vdash e \Leftarrow A$ read "under assumptions in Γ, the expression e checks against type A"

An expression e has type $A1 \cap A2$, "$A1$ intersect $A2$", if e has type $A1$ *and* e has type $A2$.

A reasonable introduction rule for $\cap$—a rule that has $\cap$ in its conclusion—is a checking rule:

$$\frac{\Gamma \vdash e \Leftarrow A1 \qquad \Gamma \vdash e \Leftarrow A2}{\Gamma \vdash e \Leftarrow (A1 \cap A2)}\ \text{Check-sect-intro}$$

If we know that an expression has type $(A1 \cap A2)$, then it has type $A1$ and also type $A2$. For example, supposing

$$\Gamma \vdash (\mathsf{Id}\ multiply) \Rightarrow \big((\mathsf{pos} * \mathsf{pos}) \to \mathsf{pos}\big) \cap \big((\mathsf{int} * \mathsf{int}) \to \mathsf{int}\big)$$

then if we apply $multiply$ to a pair of integers

$$\big(\mathsf{App}\ (\mathsf{Id}\ multiply)\ (\mathsf{Pair}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ {-3}))\big)$$

we should know from $(\mathsf{Id}\ multiply) \Rightarrow \big((\mathsf{pos} * \mathsf{pos}) \to \mathsf{pos}\big) \cap \big((\mathsf{int} * \mathsf{int}) \to \mathsf{int}\big)$ that $(\mathsf{Id}\ multiply) \Rightarrow \big((\mathsf{int} * \mathsf{int}) \to \mathsf{int}\big)$, and (using Synth-app) derive

$$\Gamma \vdash \big(\mathsf{App}\ (\mathsf{Id}\ multiply)\ (\mathsf{Pair}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ {-3}))\big) \Rightarrow \mathsf{int}$$

The step of deriving $multiply \Rightarrow A2$ from $multiply \Rightarrow (A1 \cap A2)$ is accomplished by an elimination rule, Synth-sect-elim2.

$$\frac{\Gamma \vdash e \Rightarrow (A1 \cap A2)}{\Gamma \vdash e \Rightarrow A1}\ \text{Synth-sect-elim1} \qquad \frac{\Gamma \vdash e \Rightarrow (A1 \cap A2)}{\Gamma \vdash e \Rightarrow A2}\ \text{Synth-sect-elim2}$$

Some additional rules:

$$\frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) \Rightarrow A}\ \text{Synth-var} \qquad \frac{\Gamma \vdash e1 \Rightarrow (A \to A') \qquad \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) \Rightarrow A'}\ \text{Synth-app}$$

$$\frac{n \in \mathbb{Z} \qquad n \geq 0}{\Gamma \vdash (\mathsf{Num}\ n) \Rightarrow \mathsf{pos}}\ \text{Synth-pos} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash (\mathsf{Num}\ n) \Rightarrow \mathsf{int}}\ \text{Synth-int} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \text{Check-sub}$$

Q4a [20 points] Complete the following derivation.

$$\underbrace{copy : \underbrace{(\mathsf{pos} \to (\mathsf{pos} * \mathsf{pos}))}_{Apos} \cap \underbrace{(\mathsf{int} \to (\mathsf{int} * \mathsf{int}))}_{Aint}}_{\Gamma} \vdash \big(\mathsf{App}\ (\mathsf{Id}\ copy)\ (\mathsf{Num}\ {-3})\big) \Rightarrow \underline{\qquad\qquad}$$

Question 4 [45 points]: Power of Two, continued

```
(define-type Type
  [Tpos]    ; positive integers
  [Tint]    ; integers
  [Trat]    ; rationals
  [Tbool]
  [T* (A1 Type?) (A2 Type?)]
  [T-> (domain Type?) (range Type?)]
  [Tsect (A1 Type?) (A2 Type?)])          ; A1 ∩ A2
```

Q4b [15 points] Again, the rule for checking an expression e against a given intersection type $(A1 \cap A2)$:

$$\frac{\Gamma \vdash e \Leftarrow A1 \qquad \Gamma \vdash e \Leftarrow A2}{\Gamma \vdash e \Leftarrow (A1 \cap A2)}\text{Check-sect-intro}$$

Implement Check-sect-intro in `check`, **or** use the space to (briefly) explain why this is not feasible (assuming you cannot change the signatures of `check/synth` or any existing branches of those functions). For reference, we have shown the code that implements the rule Check-pair.

$$\frac{\Gamma \vdash e1 \Leftarrow B1 \qquad \Gamma \vdash e2 \Leftarrow B2}{\Gamma \vdash (\text{Pair } e1\ e2) \Leftarrow (B1 * B2)}\text{Check-pair}$$

```
; synth : Typing-context E --> (or false Type)
; check : Typing-context E Type --> boolean
;
(define (check tc e B)
  (type-case Type B
    [Tsect (A1 A2)

      -----------------------------------------------------------

      -----------------------------------------------------------

      -----------------------------------------------------------
    ]
    [else   ; B not Tsect
        (type-case E e
          ; ...
          [Pair (e1 e2) (type-case Type B
                           [T* (B1 B2) (and (check tc e1 B1)
                                            (check tc e2 B2))]
                           [else #false])]
          ; ...
          [else #false   ; you can cross this out, if necessary

            ---------------------------------------------------------

            ---------------------------------------------------------

            ---------------------------------------------------------
          ]   ; end of else branch of type-case E e
     ]   ; end of else branch of type-case Type B
   ))
```

Question 4 [45 points]: Power of Two, continued

Q4c [10 points]

Suppose that, instead of bidirectional typing, we are using the traditional single typing judgment $\Gamma \vdash e : A$. Rules for the intersection type $\cap$ can be obtained by copying the bidirectional rules Check-sect-intro, Synth-sect-elim1 and Synth-sect-elim2, and replacing "$\Rightarrow$" and "$\Leftarrow$" with ":".

$$\frac{\Gamma \vdash e : A1 \qquad \Gamma \vdash e : A2}{\Gamma \vdash e : (A1 \cap A2)}\text{ Type-sect-intro}$$

$$\frac{\Gamma \vdash e : (A1 \cap A2)}{\Gamma \vdash e : A1}\text{ Type-sect-elim1} \qquad \frac{\Gamma \vdash e : (A1 \cap A2)}{\Gamma \vdash e : A2}\text{ Type-sect-elim2}$$

Implement Type-sect-intro in `typeof`, **or** use the space to (briefly) explain why this is not feasible (assuming you cannot change the signature of `typeof` or any existing branch).

```
(define (typeof tc e)   ; typeof : Typing-context E -> (or false Type)

    ----------------------------------------------------------

    ----------------------------------------------------------

    ----------------------------------------------------------
  (type-case E e
    ; ...
    [Pair-case (e x1 x2 eBody)
              (let ([Ae (typeof tc e)])
                (and Ae
                     (type-case Type Ae
                      [T* (A1 A2)
                           (let ([tc2 (tc/cons-tp x1 A1 (tc/cons-tp x2 A2 tc))])
                              (typeof tc2 eBody))]
                      [else #false]))))]
    ; ...
    [else #false   ; you can cross this out, if necessary

          ------------------------------------------------------

          ------------------------------------------------------

          ------------------------------------------------------
    ]))
```

Question 5 [30 points]: The Criminal Cats of West 11th Avenue

The subsumption principle states that, if $A1$ <: $A2$ (meaning that $A1$ is a subtype of $A2$), then any value of type $A1$ can safely be used wherever a value of type $A2$ is required.

Each part of this question proposes one or more subtyping rules. In each part, determine whether the rules proposed maintain the subsumption principle or violate it. If they violate the subsumption principle, give an example of an expression of type $A1$ that cannot be safely used where an expression of type $A2$ is expected.

Assume, in all the parts, that we have the following subtyping and typing rules that allow us to distinguish positive (≥ 0) rational numbers from negative (≤ 0) rational numbers through types pos and neg, which are both subtypes of rat. Also assume there is a function `print-pos` : pos $\rightarrow$ pos that can only print positive rationals, and will crash if given a rational that is less than zero.

$$\frac{}{\mathsf{pos} <: \mathsf{rat}} \qquad \frac{}{\mathsf{neg} <: \mathsf{rat}} \qquad \frac{n \geq 0}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{pos}} \qquad \frac{n \leq 0}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{neg}}$$

Example Proposed rule: $\dfrac{}{\mathsf{rat} <: \mathsf{pos}}$

Does this proposed rule: ☐ maintain the subsumption principle, or

☑ violate it (example expression: _ (Num −3) ______________)

Note: As this sample answer shows, we are **not** asking for an entire expression that will give an error when evaluated—only for an expression that (in this example) has type rat and that cannot be safely used where an expression of type pos is expected. That is, you do not have to show us an expression such as

$$(\mathsf{App}\ (\mathsf{Id}\ \texttt{print-pos})\ (\mathsf{Num}\ {-3}))$$

However, thinking about such expressions may help you answer the questions.

Q5a [15 points] Recall some relevant typing rules:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (\mathsf{Ref}\ e) : \mathsf{ref}\ A} \qquad \frac{\Gamma \vdash e : \mathsf{ref}\ A}{\Gamma \vdash (\mathsf{Deref}\ e) : A} \qquad \frac{\Gamma \vdash e1 : \mathsf{ref}\ A \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{Setref}\ e1\ e2) : A}$$

Proposed rule: $\dfrac{A <: B \qquad B <: A}{(\mathsf{ref}\ A) <: (\mathsf{ref}\ B)}$

Does this proposed rule:

☐ maintain the subsumption principle, or

☐ violate it (example expression: ________________________)

where A = ____________ and B = ____________

Q5b [15 points] The feline criminals of West 11th Avenue (perhaps confused about the meaning of "rat") have proposed that any function taking a rat as its argument can be used as a rat.

Recall some relevant typing rules:

$$\frac{\Gamma \vdash e1 : \mathsf{rat} \qquad \Gamma \vdash e2 : \mathsf{rat}}{\Gamma \vdash (\mathsf{Add}\ e1\ e2) : \mathsf{rat}} \qquad \frac{x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Lam}\ x\ eBody) : (A \rightarrow B)} \qquad \frac{\Gamma \vdash e1 : (A \rightarrow B) \qquad \Gamma \vdash e2 : B}{\Gamma \vdash (\mathsf{App}\ e1\ e2) : B}$$

Proposed subtyping rule: $\dfrac{}{(\mathsf{rat} \rightarrow B) <: \mathsf{rat}}$

Does this proposed rule: ☐ maintain the subsumption principle, or

☐ violate it (example expression: ________________________)

where B = ____________