Final Exam

CS 320, Fall 2018

Student id: _____ Name: _____

Instructions: You have 180 minutes to complete this closed-book, closed-note, closed-computer exam. Please write all answers in the provided space. Korean students should write your answers in Korean.

The solutions in this file may omit necessary details. You are not supposed to ask TAs to SOLVE the problems from scratch. When you have specific questions while solving the problems, you are more than welcome to ask them to TAs.

1) **(5pts)** Summarize what you learned from this course in one sentence.

2) **(5pts)** Write what you learned from the "Growing a Language" video lecture in one sentence.

3) **(5pts)** Suppose a garbage-collected interepreter uses the following five kinds of records:

- Tag **1**: a record containing two pointers
- Tag **2**: a record containing one pointer and one integer
- Tag **3**: a record containing one integer
- Tag **4**: a record containing one integer and one pointer
- Tag **99**: forwarding pointer (to to-space)

The interpreter has one register, which always contains a pointer, and a memory pool of size 26. The allocator/collector is a two-space copying collector, so each space is of size 13. Records are allocated consecutively in to-space, starting from the first memory location, 0.

The following is a snapshot of memory just before a collection where all memory has been allocated:

- Register: 8
- From space: 1 3 8 3 0 4 7 3 2 0 8 3 42

What are the values in the register, the from-space, and the to-space after collection? Assume that unallocated memory in to-space contains 0.

- Register: (1pt) 0
- From space: (2pts) 99 3 8 99 6 4 7 3 99 0 8 3 42
- To space: (2pts) 2 3 8 1 6 0 3 0 0 0 0 0 0

OR

- Register: (1pt) 13
- From space: (2pts) 99 16 8 99 19 4 7 3 99 13 8 3 42
- To space: (2pts) 2 16 8 1 19 13 3 0 0 0 0 0 0

4) **(5pts)** Compare programming languages with and without garbage collection. What are such programming languages? What are pros and cons?

GC: Java, Scala, $\cdots$ / safe memory management / slow

no GC: C, C++, $\cdots$ / fast / memory-related errors

5) **(5pts)** Which of the following produce different results in an eager language and a lazy language? Both produce the same result if they both produce the same number, they both produce a procedure (even if the procedure doesn't behave exactly the same when applied), or they both produce an error. Show the results of each expression in an eager language and a lazy language.

a) `{{fun {y} 12} {1 2}}`

eager: error lazy: 12

b) `{fun {x} {{fun {y} 12} {1 2}}}`

eager: closure lazy: closure

c) `{+ 1 {fun {y} 12}}`

eager: error lazy: error

d) `{+ 1 {{fun {x} {+ 1 13}} {+ 1 {fun {z} 12}}}}`

eager: error lazy: 15

e) `{+ 1 {{fun {x} {+ x 13}} {+ 1 {fun {z} 12}}}}`

eager: error lazy: error

6) **(5pts)** Consider the following `LFAE` expression:

$$
\begin{array}{rcl}
e & ::= & n \\
 & | & e + e \\
 & | & x \\
 & | & \lambda x.\ e \\
 & | & e\ e
\end{array}
\qquad
\begin{array}{rcccl}
n & \in & \text{Num} & & \\
\sigma & \in & \text{Env} & = & \text{Var} \xrightarrow{\text{fin}} \text{Val} \\
x & \in & \text{Var} & & \\
\langle \lambda x.e,\ \sigma \rangle & \in & \text{Clo} & = & \text{Var} \times \text{Expr} \times \text{Env} \\
v & \in & \text{Val} & = & \mathbb{Z} + \text{Clo} + \text{ExprV} \\
(e, \sigma) & \in & \text{ExprV} & = & \text{Expr} \times \text{Env}
\end{array}
$$

with the following "strict" evaluation of the form $\boxed{v \Downarrow v}$:

$$
n \Downarrow n \qquad \langle \lambda x.e,\ \sigma \rangle \Downarrow \langle \lambda x.e,\ \sigma \rangle \qquad \frac{\sigma \vdash e \Rightarrow v_1 \qquad v_1 \Downarrow v_2}{(e, \sigma) \Downarrow v_2}
$$

Write the operational semantics of `LFAE`, where a value is strictly evaluted when it is used, that is, strict evaluation is deferred as much as possible. Thus, the evaluation of "$\lambda x.x\ y$" is not an error.

– n

$$\sigma \vdash n \Rightarrow n$$

– $e + e$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

– x

$$\frac{x \in \text{Dom}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}$$

– $\lambda x.\ e$

$$\sigma \vdash \lambda x.\ e \Rightarrow \langle \lambda x.e,\ \sigma \rangle$$

– $e\ e$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow \langle \lambda x.e,\ \sigma' \rangle \qquad \sigma'[x \mapsto (e_2, \sigma)] \vdash e \Rightarrow v_2}{\sigma \vdash e_1\ e_2 \Rightarrow v_2}$$

7) **(5pts)** Write the typing rule for the following expression:

`{with {`$x\ \tau\ e_1$`}` e_2`}`

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash e_1 : \tau \quad \Gamma[x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \{\texttt{with}\ \{x\ \tau\ e_1\}\ e_2\} : \tau'}$$

8) **(5pts)** Consider the following partial implementation of the `compile` function:

```
type CEnv = List[String]

def compile(fae: FAE, cenv: CEnv): CFAE = fae match {
  case Num(n) => CNum(n)
  case Add(l, r) => CAdd(compile(l, cenv), compile(r, cenv))
  case Sub(l, r) => CSub(compile(l, cenv), compile(r, cenv))
  case Id(name) => CId(locate(name, cenv))
  case Fun(param, body) => CFun(compile(body, param :: cenv))
  case App(l, r) => CApp(compile(l, cenv), compile(r, cenv))
}

def locate(name: String, cenv: CEnv): Int = cenv match {
  case Nil  => error("a free identifier: $name")
  case h::t => if (name == h) 0 else locate(name, t) + 1
}
```

What is the result of calling the `compile` function with the following expression:

```
{{{{fun {x} {fun {y} {fun {z} {+ x {+ y z}}}}} 8} 42} 320}
```

```
CApp(CApp(CApp(CFun(CFun(CFun(CAdd(CId(2), CAdd(CId(1), CId(0)))))),
               CNum(8)),
          CNum(42)),
     CNum(320))
```

9) **(10pts)** Consider the following grammar:

e	::=	n	number
	|	x	identifier
	|	e `+` e	addition of two numbers
	|	`ref` e	location construction with a given initial value
	|	`!` e	dereferencing
	|	e `:=` e	assignment evaluating the right-hand side (RHS) first, producing the value of RHS
	|	e `;` e	sequencing
	|	`let` x `=` e `in` e	name binding
v	::=	n	number
	|	l	location
τ	::=	`num`	type of numbers
	|	`ref` τ	type of locations that contain values of type τ

$x \in Id$
$l \in Loc$
$v \in Val$
$\sigma \in Id \to Loc$
$M \in Loc \to Val$

where evaluation of "`let` x `=` 42 `in` x" results in 42.

a) **(5pts)** Write the operational semantics of the form $\boxed{\sigma, M \vdash e \Rightarrow v, M}$ for the expressions with the `BFAE` semantics.

$$\sigma, M \vdash n \Rightarrow n, M$$

$$\sigma, M \vdash x \Rightarrow M(\sigma(x)), M$$

$$\frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2, M_2}$$

$$\frac{\sigma, M \vdash e \Rightarrow v, M' \quad l \notin Dom(M')}{\sigma, M \vdash \texttt{ref } e \Rightarrow l, M'[l \mapsto v]}$$

$$\frac{\sigma, M \vdash e \Rightarrow l, M'}{\sigma, M \vdash \texttt{ ! } e \Rightarrow M'(l), M'}$$

$$\frac{\sigma, M \vdash e_2 \Rightarrow v, M_1 \quad \sigma, M_1 \vdash e_1 \Rightarrow l, M_2}{\sigma, M \vdash e_1 \texttt{ := } e_2 \Rightarrow v, M_2[l \mapsto v]}$$

$$\frac{\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1 \texttt{ ; } e_2 \Rightarrow v_2, M_2}$$

$$\frac{\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \quad \sigma[x \mapsto l], M_1[l \mapsto v_1] \vdash e_2 \Rightarrow v_2, M_2 \quad l \notin Dom(M_1)}{\sigma, M \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Rightarrow v_2, M_2}$$

b) **(5pts)** Write the typing rules of the form $\boxed{\Gamma \vdash e : \tau}$ for the expressions. Assignments should not change the types of the values at a given location.

$$\Gamma \vdash n : \texttt{num}$$

$$\frac{x \in Dom(\Gamma)}{\Gamma \vdash x : \ \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma_1 \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 \ \texttt{+} \ e_2 : \texttt{num}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref} \ e : \texttt{ref} \ \tau}$$

$$\frac{\Gamma \vdash e : \texttt{ref} \ \tau}{\Gamma \vdash \texttt{!} \ e : \tau}$$

$$\frac{\Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_1 : \texttt{ref} \ \tau}{\Gamma \vdash e_1 \ \texttt{:=} \ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ \texttt{;} \ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 : \tau_2}$$

10) (10pts) Consider the following `KCFAE` expression:

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}l}
e & ::= & n & \kappa & ::= & [\Box] & \texttt{(mkK)} \\
 & | & e + e & & | & [\Box + (e, \sigma)] :: \kappa & \texttt{(addSecondK e ds k)} \\
 & | & x & & | & [v + \Box] :: \kappa & \texttt{(doAddK v k)} \\
 & | & \lambda x.\ e & & | & [\Box\ (e, \sigma)] :: \kappa & \texttt{(appArgK e ds k)} \\
 & | & e\ e & & | & [v\ \Box] :: \kappa & \texttt{(doAppK v k)} \\
 & | & \texttt{withcc}\ x\ e & & & & \\
\end{array}
$$

$$
\begin{array}{rcl}
e \in & \text{Expr} & \\
\sigma \in & \text{Env} & = \text{Var} \xrightarrow{\text{fin}} \text{Val} \\
x \in & \text{Var} & \\
\langle \lambda x.e,\ \sigma \rangle \in & \text{Clo} & = \text{Var} \times \text{Expr} \times \text{Env} \\
v \in & \text{Val} & = \mathbb{Z} + \text{Clo} + \text{Cont} \\
\kappa \in & \text{Cont} &
\end{array}
$$

Write the big-step operational semantics of `KCFAE` of the form $\boxed{\sigma, \kappa \vdash e \Rightarrow v}$ using the continue semantics of the form $\boxed{v_1 \mapsto \kappa \Downarrow v_2}$, which denotes that v_2 is the result of continuation κ taking the value v_1.

a) $\boxed{v \mapsto \kappa \Downarrow v}$

* $[\Box]$

$$v \mapsto [\Box] \Downarrow v$$

* $[\Box + (e, \sigma)] :: \kappa$

$$\frac{\sigma, [v_1 + \Box] :: \kappa \vdash e \Rightarrow v}{v_1 \mapsto [\Box + (e, \sigma)] :: \kappa \Downarrow v}$$

* $[v + \Box] :: \kappa$ **(1pt)**

$$\frac{n_1 + n_2 \mapsto \kappa \Downarrow v}{n_2 \mapsto [n_1 + \Box] :: \kappa \Downarrow v}$$

* $[\Box\ (e, \sigma)] :: \kappa$ **(1pt)**

$$\frac{\sigma, [v\ \Box] :: \kappa \vdash e \Rightarrow v'}{v \mapsto [\Box\ (e, \sigma)] :: \kappa \Downarrow v'}$$

* $[v\ \Box] :: \kappa$ **(1pt + 2pts)**

$$\frac{\sigma[x \mapsto v_2], \kappa \vdash e \Rightarrow v}{v_2 \mapsto [\langle \lambda x.e,\ \sigma \rangle\ \Box] :: \kappa \Downarrow v}$$

$$\frac{v_2 \mapsto \kappa' \Downarrow v}{v_2 \mapsto [\kappa'\ \Box] :: \kappa \Downarrow v}$$

b) $\boxed{\sigma, \kappa \vdash e \Rightarrow v}$

* n

$$\frac{n \mapsto \kappa \Downarrow v}{\sigma, \kappa \vdash n \Rightarrow v}$$

* $e + e$

$$\frac{\sigma, [\square + (e_2, \sigma)] :: \kappa \vdash e_1 \Rightarrow v}{\sigma, \kappa \vdash e_1 + e_2 \Rightarrow v}$$

* x

$$\frac{x \in \mathrm{Dom}(\sigma) \qquad \sigma(x) \mapsto \kappa \Downarrow v}{\sigma, \kappa \vdash x \Rightarrow v}$$

* $\lambda x.\ e$

$$\frac{\langle \lambda x.e,\ \sigma \rangle \mapsto \kappa \Downarrow v}{\sigma, \kappa \vdash \lambda x.\ e \Rightarrow v}$$

* $e\ e$

$$\frac{\sigma, [\square\ (e_2, \sigma)] :: \kappa \vdash e_1 \Rightarrow v}{\sigma, \kappa \vdash e_1\ e_2 \Rightarrow v}$$

* `withcc` $x\ e$

$$\frac{\sigma[x \mapsto \kappa], \kappa \vdash e \Rightarrow v}{\sigma, \kappa \vdash \texttt{withcc}\ x\ e \Rightarrow v}$$

11) (10pts) Consider the following KCFAE expression:

$$
\begin{array}{rcl@{\quad}rcl@{\quad}l@{\quad}rcl}
e & ::= & n & \kappa & ::= & \square & & n \in & \mathbb{Z} & v \in \text{Val} = \mathbb{Z} + \text{Clo} + \text{Continuation} \\
 & | & e + e & & | & \sigma \vdash e :: \kappa & \text{interpretation} & x \in & \text{Var} & \langle \lambda x.e,\ \sigma \rangle \in \text{Clo} = \text{Var} \times \text{Expr} \times \text{Env} \\
 & | & x & & | & \texttt{(+)} :: \kappa & \text{addition} & e \in & \text{Expr} & \sigma \in \text{Env} = \text{Var} \xrightarrow{\text{fin}} \text{Val} \\
 & | & \lambda x.\ e & & | & \texttt{(@)} :: \kappa & \text{application} & \kappa \in & \text{Cont} & \langle \kappa, s \rangle \in \text{Continuation} = \text{Cont} \times \text{Stack} \\
 & | & e\ e & s & ::= & \blacksquare & & s \in & \text{Stack} & \\
 & | & \texttt{withcc}\ x\ e & & | & v :: s & &
\end{array}
$$

a) Fill in the missing five cases of the small-step operational semantics of KCFAE of the form $\boxed{\kappa \mid\mid s \ \rightarrow\ \kappa \mid\mid s}$.

$$\sigma \vdash n :: \kappa \mid\mid s \ \rightarrow\ \kappa \mid\mid n :: s$$

$$\sigma \vdash e_1 \texttt{ + } e_2 :: \kappa \mid\mid s \ \rightarrow\ \sigma \vdash e_1 :: \sigma \vdash e_2 :: \texttt{(+)} :: \kappa \mid\mid s$$

$$\texttt{(+)} :: \kappa \mid\mid n_2 :: n_1 :: s \ \rightarrow\ \kappa \mid\mid n_1 + n_2 :: s$$

$$\sigma \vdash x :: \kappa \mid\mid s \ \rightarrow\ \kappa \mid\mid \sigma(x) :: s$$

$$\sigma \vdash \lambda x.\, e :: \kappa \mid\mid s \ \rightarrow\ \kappa \mid\mid \langle \lambda x.\, e, \sigma \rangle :: s$$

$$\sigma \vdash e_1\ e_2 :: \kappa \mid\mid s \ \rightarrow\ \sigma \vdash e_1 :: \sigma \vdash e_2 :: \texttt{(@)} :: \kappa \mid\mid s$$

$$\texttt{(@)} :: \kappa \mid\mid v :: \langle \lambda x.\, e, \sigma \rangle :: s \ \rightarrow\ \sigma[x \mapsto v] \vdash e :: \kappa \mid\mid s$$

$$\texttt{(@)} :: \kappa \mid\mid v :: \langle \kappa', s' \rangle :: s \ \rightarrow\ \kappa' \mid\mid v :: s'$$

$$\sigma \vdash \texttt{withcc}\ x\ e :: \kappa \mid\mid s \ \rightarrow\ \sigma[x \mapsto \langle \kappa, s \rangle] \vdash e :: \kappa \mid\mid s$$

b) Show the interpretation steps of the expression (`withcc` x (42 `+` (x 2))) `+` 8 where the value of a parenthesized expression (e) is the value of e.

$$
\begin{array}{lll}
 & \varnothing \vdash (\texttt{withcc}\ x\ (42\ \texttt{+}\ (x\ 2)))\ \texttt{+}\ 8 :: \square & \| \ \blacksquare \\
\equiv & \varnothing \vdash e_1\ \texttt{+}\ 8 :: \square & \| \ \blacksquare \\
\rightarrow & \varnothing \vdash e_1 :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ \blacksquare \\
\equiv & \varnothing \vdash \texttt{withcc}\ x\ e_2 :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ \blacksquare \\
\rightarrow & \sigma_1 \vdash e_2 :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ \blacksquare \\
\equiv & \sigma_1 \vdash 42\ \texttt{+}\ (x\ 2) :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ \blacksquare \\
\rightarrow & \sigma_1 \vdash 42 :: \sigma_1 \vdash (x\ 2) :: \texttt{(+)} :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ \blacksquare \\
\rightarrow & \sigma_1 \vdash (x\ 2) :: \texttt{(+)} :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ 42 :: \blacksquare \\
\rightarrow & \sigma_1 \vdash x :: \sigma_1 \vdash 2 :: \texttt{(@)} :: \texttt{(+)} :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ 42 :: \blacksquare \\
\rightarrow & \sigma_1 \vdash 2 :: \texttt{(@)} :: \texttt{(+)} :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ \langle \varnothing \vdash 8 :: \texttt{(+)} :: \square, \blacksquare \rangle :: 42 :: \blacksquare \\
\rightarrow & \texttt{(@)} :: \texttt{(+)} :: \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ 2 :: \langle \varnothing \vdash 8 :: \texttt{(+)} :: \square, \blacksquare \rangle :: 42 :: \blacksquare \\
\rightarrow & \varnothing \vdash 8 :: \texttt{(+)} :: \square & \| \ 2 :: \blacksquare \\
\rightarrow & \texttt{(+)} :: \square & \| \ 8 :: 2 :: \blacksquare \\
\rightarrow & \square & \| \ 10 :: \blacksquare
\end{array}
$$

$$
\begin{array}{rcl}
 & e_0 & = \ e_1\ \texttt{+}\ 8 \\
 & e_1 & = \ \texttt{withcc}\ x\ e_2 \\
\text{where} & e_2 & = \ 42\ \texttt{+}\ e_3 \\
 & e_3 & = \ (x\ 2) \\
 & \sigma_1 & = \ [x \mapsto \langle \varnothing \vdash 8 :: \texttt{(+)} :: \square, \blacksquare \rangle]
\end{array}
$$

12) **(5pts)** Rewrite the following code using explicit annotation of polymorphic types with `tyfun` and `@` to replace all the occurrences of `?` with types and to make function calls to take explicit type arguments.

$$\frac{\Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \texttt{[tyfun [}\alpha\texttt{] }e\texttt{]} : (\forall \alpha\ \tau)} \qquad \frac{\Gamma \vdash \tau_0 \quad \Gamma \vdash e : (\forall \alpha\ \tau_1)}{\Gamma \vdash \texttt{[@ }e\ \tau_0\texttt{]} : \tau_1[\alpha \leftarrow \tau_0]}$$

$$\frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash (\forall \alpha\ \tau)} \qquad [\ldots \alpha \ldots] \vdash \alpha$$

```
{with {f : ? {fun {g : ?} {fun {v : ?} {g v}}}}
      {with {g : ? {fun {x : ?} x}}
            {{f g} 10}}}
```

```
{with {f : (∀α (∀β ((α → β) → (α → β))))
           [tyfun [α] [tyfun [β]
               {fun {g : (α → β)} {fun {v : α} {g v}}}]]}
      {with {g : (∀α (α → α))
                 [tyfun [α] {fun {x : α} x}]}
            {{[@ [@ f num] num] [@ g num]} 10}}}
```

f's type 1pt, f's body 1pt, g's type 1pt, g's body 1pt, app 1pt

13) **(15pts)** We'd like to allow the following expression:

```
{withtype {{alpha list} {empty num}
                        {cons (alpha * {alpha list})}}
          {rec {len : (forall alpha ({alpha list} -> num))
                    [tyfun [alpha]
                           {fun {l : {alpha list}}
                                {cases {alpha list} l
                                  {empty {n} 0}
                                  {cons {fxr}
                                        {+ 1 {len {snd fxr}}}}}}]}
               {+ {[@ len num] {[@ cons num] {pair 1 {[@ empty num] 0}}}}
                  {[@ len (num -> num)] {[@ empty (num -> num)] 0}}}}}
```

and of course the following as well:

```
{withtype {fruit {banana num} {apple bool}}
          {{fun {x : fruit} {cases fruit x {banana {n} {+ n 9}} {apple  {b} 76}}}
           {banana 2}}}
```

Thus, we define the following language:

e ::= n
| {+ e e}
| x
| {fun {x:τ} e}
| {e e}
| {rec {x:τ e} e}
| {pair e e}
| {fst e}
| {snd e}
| {withtype {γ {x τ} {x τ}} e}
| {cases γ e {x {x} e} {x {x} e}}
| [tyfun [α] e]
| [@ e τ]

τ ::= num
| ($\tau \to \tau$)
| ($\tau \times \tau$)
| γ
| α
| ($\forall\alpha$ τ)

γ ::= {α t}
| t

a) **(4pts)** Write the operational semantics of the form $\boxed{\sigma \vdash e \Rightarrow v}$ for the expressions `withtype`, `cases`, and `{`e e`}`.

(4pts) 1pt for each rule

$$\frac{\sigma[x_1 \mapsto \texttt{cV(1)},\ x_2 \mapsto \texttt{cV(2)}] \vdash e \Rightarrow v}{\sigma \vdash \{\texttt{withtype}\ \{\gamma\ \{x_1\ \tau_1\}\ \{x_2\ \tau_2\}\}\ e\} \Rightarrow v}$$

$$\frac{\sigma \vdash e_1 \Rightarrow \texttt{cV(}i\texttt{)} \quad \sigma \vdash e_2 \Rightarrow v \quad i = 1 \text{ or } 2}{\sigma \vdash \{e_1\ e_2\} \Rightarrow \texttt{vV(}i\texttt{,}v\texttt{)}}$$

$$\frac{\sigma \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle \quad \sigma \vdash e_2 \Rightarrow v \quad \sigma'[x \mapsto v] \vdash e \Rightarrow v'}{\sigma \vdash \{e_1\ e_2\} \Rightarrow v'}$$

$$\frac{\sigma \vdash e_0 \Rightarrow \texttt{vV(}i\texttt{,}v\texttt{)} \quad \sigma[y_i \mapsto v] \vdash e_i \Rightarrow v' \quad i = 1 \text{ or } 2}{\sigma \vdash \{\texttt{cases}\ \gamma\ e_0\ \{x_1\ \{y_1\}\ e_1\}\ \{x_2\ \{y_2\}\ e_2\}\} \Rightarrow v'}$$

b) **(6pts)** Write the typing rules of the form $\boxed{\Gamma \vdash e : \tau}$ for the expressions `withtype` and `cases` and the well-formedness of user-defined types.

(6pts) 1 / 2 / 2 / 1

$$\frac{\begin{array}{c}\Gamma' = \Gamma[t = x_1@\tau_1 + x_2@\tau_2, x_1 : (\tau_1 \to t), x_2 : (\tau_2 \to t)] \\ \Gamma' \vdash \tau_1 \quad \Gamma' \vdash \tau_2 \quad \Gamma' \vdash e : \tau_0 \quad t \text{ not in } \tau_0\end{array}}{\Gamma \vdash \{\texttt{withtype}\ \{t\ \{x_1\ \tau_1\}\ \{x_2\ \tau_2\}\}\ e\} : \tau_0}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[\{\alpha\ t\} = x_1@\tau_1 + x_2@\tau_2, x_1 : (\forall\alpha\ (\tau_1 \to \{\alpha\ t\})), x_2 : (\forall\alpha\ (\tau_2 \to \{\alpha\ t\})), \alpha] \\ \Gamma' \vdash \tau_1 \quad \Gamma' \vdash \tau_2 \quad \Gamma' \vdash e : \tau_0 \quad \{\alpha\ t\} \text{ not in } \tau_0\end{array}}{\Gamma \vdash \{\texttt{withtype}\ \{\{\alpha\ t\}\ \{x_1\ \tau_1\}\ \{x_2\ \tau_2\}\}\ e\} : \tau_0}$$

$$\frac{\begin{array}{c}\Gamma = [\ldots \gamma = x_1@\tau_1 + x_2@\tau_2 \ldots] \\ \Gamma \vdash e_0 : \gamma \quad \Gamma[y_1 : \tau_1] \vdash e_1 : \tau_0 \quad \Gamma[y_2 : \tau_2] \vdash e_2 : \tau_0\end{array}}{\Gamma \vdash \{\texttt{cases}\ \gamma\ e_0\ \{x_1\ \{y_1\}\ e_1\}\ \{x_2\ \{y_2\}\ e_2\}\} : \tau_0}$$

$[\ldots \{\alpha\ t\} = \ldots] \vdash \{\alpha\ t\}$

$[\ldots t = \ldots] \vdash t$

c) **(5pts)** Consider the following expression:

```
{withtype {fruit {apple num} {banana num}}
          {{withtype {color {apple num} {banana num}}
                     {fun {x : fruit}
                          {cases fruit x
                            {apple  {a} {apple  {+ a 1}}}
                            {banana {b} {banana {+ b 1}}}}}}}}
           {apple 42}}}
```

If the result of type checking the expression is `fruit`, explain why in detail. Otherwise, revise the typing rules to make it `fruit`.

$$\frac{\Gamma = [\cdots\ t = x_1@\tau_1 + x_2@\tau_2 \cdots] \qquad \Gamma' = \Gamma[x_1 : (\tau_1 \to t),\ x_2 : (\tau_2 \to t)] \qquad \Gamma \vdash e_0 : t \qquad \Gamma'[y_1 : \tau_1] \vdash e_1 : \tau \qquad \Gamma'[y_2 : \tau_2] \vdash e_2 : \tau}{\Gamma \vdash \{\texttt{cases}\ t\ e_0\ \{x_1\ \{y_1\}\ e_1\}\ \{x_2\ \{y_2\}\ e_2\}\} : \tau}$$

14) (10pts) Consider the following language:

class declaration	d	::= `class` C `extends` C `{` g^* k m^* `}`
field declaration	g	::= x`:`C`;`
constructor declaration	k	::= C`(`(x`:`C)*`) { super(`x^*`); (this.`x `=` x`;)`* `}`
method declaration	m	::= f`(`(x`:`C)*`):`C `{ return` e`; }`
expression	e	::= x | e`.`x | e`.`f`(`e^*`)` | `new` C`(`e^*`)`

where metavariables C, D, and E range over class names, x ranges over field names, and f ranges over method names. We assume that the set of variables includes the special variable `this`, which cannot be used as the name of an argument to a method. Instead, it is implicitly bound in every method declaration. The evaluation rule for method invocation will substitute an appropriate object for `this`, in addition to substituting the argument values for the parameters.

With the following subtyping rules:

$$C <: C \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{\texttt{class } C \texttt{ extends } D \{\cdots\}}{C <: D}$$

and the following helper functions:

$\mathit{fields}(C) = x_1 : C_1 \cdots x_n : C_n$	the fields of class C are x_1 to x_n of types C_1 to C_n, respectively
$\mathit{mtype}(C, f) = C_1 \cdots C_n \to C_{n+1}$	the type of method f defined in class C is $C_1 \cdots C_n \to C_{n+1}$
$\mathit{ctype}(C) = C_1 \cdots C_n \to C$	the type of the constructor of class C is $C_1 \cdots C_n \to C$

a) Write the typing rules of the form $\boxed{\Gamma \vdash e : C}$ for the expressions. The typing rules for constructors and method invocations check that each actual argument has a type that is a subtype of the corresponding formal parameter type.

$$\frac{x \in \mathit{Dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e : C \qquad \mathit{fields}(C) = x_1 : C_1 \cdots x_n : C_n}{\Gamma \vdash e.x_i : C_i}$$

$$\frac{\Gamma \vdash e : C \qquad \mathit{mtype}(C, f) = C_1 \cdots C_n \to C_{n+1} \qquad \Gamma \vdash e_1 : D_1 \quad \cdots \quad \Gamma \vdash e_n : D_n \qquad D_1 <: C_1 \quad \cdots \quad D_n <: C_n}{\Gamma \vdash e.f(e_1, \cdots e_n) : C_{n+1}}$$

$$\frac{\mathit{ctype}(C) = C_1 \cdots C_n \to C \qquad \Gamma \vdash e_1 : D_1 \quad \cdots \quad \Gamma \vdash e_n : D_n \qquad D_1 <: C_1 \quad \cdots \quad D_n <: C_n}{\Gamma \vdash \texttt{new } C(e_1, \cdots e_n) : C}$$

b) The following judgment states that:

$$C \ \vdash \ f\texttt{(}x_1\texttt{:}C_1,\ \cdots\ x_n\texttt{:}C_n\texttt{):}C_{n+1}\ \texttt{\{ return } e\texttt{; \}}$$

the method declaration defined in C is well typed. Write its typing rule which checks the following:

* The type of f's body expression e should be a subtype of the annotated return type C_{n+1}.
* When class C explicitly extends D ($\texttt{class } C \texttt{ extends } D \texttt{ \{ } \cdots \texttt{ \}}$), if D defines any method of name f, the type of f in D should be the same with the type of f in C.

$$[x_1 : C_1 \ \cdots \ x_n : C_n \ \texttt{this} : C] \vdash e : C' \qquad C' <: C_{n+1}$$

$$\texttt{class } C \texttt{ extends } D \ \{\cdots\}$$

$$\frac{mtype(D, f) = C'_1 \cdots C'_{n'} \rightarrow C'_{n'+1} \Rightarrow \ n = n' \wedge C_1 = C'_1 \wedge \cdots C_{n+1} = C'_{n+1}}{C \ \vdash \ f\texttt{(}x_1\texttt{:}C_1,\ \cdots\ x_n\texttt{:}C_n\texttt{):}C_{n+1}\texttt{\{ return } e\texttt{; \}}}$$