Lenguajes de Programación 2020-1
Facultad de Ciencias UNAM
Tarea Examen Parcial 4

Sandra del Mar Soto Corderi   Edgar Quiroz Castañeda

Fecha de entrega: 2 de diciembre de 2019

Esta tarea vale 8 puntos sobre el parcial 4, la calificación se completa con una pregunta presencial el 29 de noviembre de 2019

1. (1pt) Se desea implementar una función `ct` que reciba un árbol heterogéneo de naturales o booleanos y devuelva la conjunción de sus elementos siempre y cuando todos sean booleanos y en otro caso devuelva el valor `n+1`, donde `n` es el primer natural encontrado en el árbol. Para este propósito defina una función `ctaux` y una expresión `e` tal que la función `ct` quede implementada como

```
ct t = handle ctaux t with x => e
```

Bosqueje la evaluación de la expresión

```
Node (iszero 9) (Node False Void Void) (Node 5 Void Void)
```

en la máquina $\mathcal{K}$.

Puede omitir varios pasos pero no los que involucren el manejo de excepciones.

Sugerencia: Es más fácil si define `ctaux` a partir de una función binaria `vand` que realice la conjunción si sus argumentos son booleanos y en caso contrario lance una excepción adecuada de forma que sea manejada por `e`. Puede suponer que existe una función unaria `isbool` que verifica si su argumento es o no un booleano.

Solución:

Definiendo `vand`.

```
-- Función auxiliar para mandar excepciones
bool_panic p = let x = p in if isbool x then x else raise(x)

-- Realiza la conjunción si sus argumentos son booleanos o lanza una
-- excepción con el primer argumeto que no lo sea.
vand p q = and (bool_panic p) (bool_panic q)
```

Ahora, hay que usar esta función para definir `ctaux`.

```
ctaux Void = True
ctaux (Node r i d) = vand r (vand (ctaux i) (ctaux d))
```

Y usando esto, se define `ct` como

```
ct t = handle ctaux t with x => x + 1
```

Se definen algunos sinónimos para hacer más corta la evaluación

```
-- constantes
f = False
t = True
-- arboles
i = (Node False Void Void)
d = (Node 5 Void Void)
```

```
a = (Node (iszero 9) i d)
-- expresiones booleanas simples
p1 = iszero 9
-- pilas
s1 = and(-, e2), handle(-, x.x+1)
s2 = and(f, -), handle(-, x.x+1)
s3 = and(-, e7), let(-, x.e10), s2
s4 = and(f, -), let(-, e10), s3
-- otras expresiones
e3 = (vand (ctaux i) (ctaux d))
e2 = bool-panic e3
e4 = ctaux i
e5 = (bool-panic (ctaux i))
e6 = ctaux d
e7 = (bool-panic (ctaux d))
e11 = ctaux Void
e12 = bool-panic e11
e8 = (vand e11 e11)
e9 = bool-panic e8
e10 = if (isbool x) then x else raise(x)
```

Ahora, vamos a evaluar la expresión:

Ahora, para evaluar la expresión, es útil definir lo siguiente

```
a = Node (iszero 9) (Node False Void Void) (Node 5 Void Void)
r = iszero 9
i = Node False Void Void
d = Node 5 Void Void
```

Evaluamos:

$$
\begin{array}{ll}
 & \Box \succ \texttt{ct a} \\
\to_\beta & \Box \succ \texttt{handle ctaux a with x =>x+1} \\
\to_\mathcal{K} & \texttt{handle(-, x.x+1)} \succ \texttt{ctaux a} \\
\to_\beta & \texttt{handle(-, x.x+1)} \succ \texttt{and (bool-panic p1) e2} \\
\to_\mathcal{K} & \texttt{and(-, e2), handle(-, x.x+1)} \succ \texttt{bool-panic p1} \\
\to_\beta & \texttt{s1} \succ \texttt{let x = p1 in e10} \\
\to_\mathcal{K} & \texttt{let(-, x.e10), s1} \succ \texttt{iszero 9} \\
\to_\mathcal{K}^\star & \texttt{let(-, e10), s1} \prec \texttt{f} \\
\to_\mathcal{K} & \texttt{s1} \succ \texttt{if isbool x then x else raise(x) [x := f]} \\
\to_\mathcal{K}^\star & \texttt{and(-, e2), handle(-, x.x+1)} \prec \texttt{f} \\
\to_\mathcal{K} & \texttt{and(f, -), handle(-, x.x+1)} \succ \texttt{bool-panic e3} \\
\to_\beta & \texttt{s2} \succ \texttt{let x = e3 in e10} \\
\to_\mathcal{K} & \texttt{let(-, x.e10), s2} \succ \texttt{e3} \\
\to_\beta & \texttt{let(-, x.e10), s2} \succ \texttt{and e5 e7} \\
\to_\mathcal{K} & \texttt{and(-, e7), let(-, x.e10), s2} \succ \texttt{e5} \\
\to_\beta & \texttt{s3} \succ \texttt{let x = e4 in e10} \\
\to_\mathcal{K} & \texttt{let(-, e10), s3} \succ \texttt{e4}
\end{array}
$$

$\to_{\beta}$ let(-, e10), s3 $\succ$ and (bool-panic f) e9
$\to_{\mathcal{K}}$ and(-, e9), let(-, e10), s3 $\succ$ bool-panic f
$\to_{\beta}$ and(-, e9), let(-, e10), s3 $\succ$ let x = f in e10
$\to_{\mathcal{K}}^{\star}$ and(-, e9), let(-, e10), s3 $\prec$ f
$\to_{\mathcal{K}}$ and(f, -), let(-, e10), s3 $\succ$ e9
$\to_{\beta}$ s4 $\succ$ let x = e8 in e10
$\to_{\mathcal{K}}$ let(-, x.e10), s4 $\succ$ e8
$\to_{\beta}$ let(-, x.e10), s4 $\succ$ and e12 e12
$\to_{\mathcal{K}}$ and(-, e12), let(-, x.e10), s4 $\succ$ e12
$\to_{\beta}$ and(-, e12), let(-, x.e10), s4 $\succ$ let x = e11 in e10
$\to_{\mathcal{K}}$ let(, x.e10), and(-, e12), let(-, x.e10), s4 $\succ$ ctaux Void
$\to_{\mathcal{K}}$ let(, x.e10), and(-, e12), let(-, x.e10), s4 $\prec$ t
$\to_{\mathcal{K}}$ and(-, e12), let(-, x.e10), s4 $\succ$ (if isbool x then x else raise(x))[x := true]
$\to_{\mathcal{K}}^{\star}$ and(true, -), let(-, x.e10), s4 $\succ$ e12
$\to_{\mathcal{K}}^{\star}$ and(true, -), let(-, x.e10), s4 $\prec$ t
$\to_{\mathcal{K}}^{\star}$ s4 $\succ$ (if isbool x then x else raise(x))[x := t]
$\to_{\mathcal{K}}^{\star}$ and(f, -), let(-, e10), s3 $\prec$ t
$\to_{\mathcal{K}}^{\star}$ s3 $\succ$ (if isbool x then x else raise(x))[x := f]
$\to_{\mathcal{K}}^{\star}$ and(-, e7), let(-, x.e10), s2 $\prec$ f
$\to_{\mathcal{K}}^{\star}$ and(f, -), let(-, x.e10), s2 $\succ$ e7
$\to_{\beta}$ and(f, -), let(-, x.e10), s2 $\succ$ let x = e6 in e10
$\to_{\mathcal{K}}$ let(-, x.e10), and(f, -), let(-, x.e10), s2 $\succ$ e6
$\to_{\beta}$ let(-, x.e10), and(f, -), let(-, x.e10), s2 $\succ$ and (bool-panic 5) e8
$\to_{\mathcal{K}}$ and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 $\succ$ bool-panic 5
$\to_{\beta}$ and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 $\succ$ let x = 5 in e10
$\to_{\mathcal{K}}^{\star}$ and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 $\succ$ if isbool 5 then 5 else raise(5)
$\to_{\mathcal{K}}^{\star}$ and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 $\ll$ raise(5)
$\to_{\mathcal{K}}^{\star}$ and(f, -), let(-, x.e10), s2 $\ll$ raise(5)
$\to_{\mathcal{K}}^{\star}$ and(f, -), handle(-, x.x+1) $\ll$ raise(5)
$\to_{\mathcal{K}}^{\star}$ handle(-, x.x+1) $\ll$ raise(5)
$\to_{\mathcal{K}}^{\star}$ $\square \succ$ (x+1)[x := 5]
$\to_{\mathcal{K}}^{\star}$ $\square \prec$ 6

2. (1pt) Considere el siguiente programa, donde suponemos que el lenguaje contiene un operador primitivo `not` para la negación booleana.

```
not letcc k2 in
    iszero(2 + letcc k1 in
        3  + if x = pred 8 then
            4 * pred (continue k1 6
            else 5 * suc (continue k2 false)
        end
        )
    end
end
```

a) ¿Cuáles son los tipos de `k1` y `k2`?

`k1 : Cont(Integer)   k2 : Cont(Bool)`

b) ¿A qué continuaciones se ligan las variables `k1` y `k2`?

k1 -->(continue k1 6) $\rightarrow_{\mathcal{K}}$ (continue (not(-), iszero(-), suma(2, -)), 6)

k2 -->(continue k2 false) $\rightarrow_{\mathcal{K}}$ (continue (not(-)), false)

c) ¿A qué se evalúa el programa para x = 7 y para $x \neq 7$?
Evaluamos, considerando x = 7

$\Box \succ$ not(letcc[Bool](k2.iszero(2 + letcc[Integer](k1.3 +
if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(k2,False))))))

$\rightarrow_{\mathcal{K}}$ not(-); $\Box \succ$ letcc[Bool](k2.iszero(2 + letcc[Integer](k1.3 +
if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(k2,False)))))

$\rightarrow_{\mathcal{K}}$ not(-); $\Box \succ$ iszero(2 + letcc[Integer](k1.3 +
if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(k2,False)))) [k2:= cont(not(-); $\Box$)]

$\equiv$ not(-); $\Box \succ$ iszero(2 + letcc[Integer](k1.3 +
if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); $\Box$),False))))

$\rightarrow_{\mathcal{K}}$ iszero(-); not(-); $\Box \succ$ 2 + letcc[Integer](k1.3 +
if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); $\Box$),False)))

$\rightarrow_{\mathcal{K}}$ suma(-, letcc[Integer](k1.3 + if(x=pred(8),4 * pred(continue(k1,6)), 5 *
suc(continue(cont(not(-); $\Box$),False)))); iszero(-); not(-); $\Box \succ$ 2

$\rightarrow_{\mathcal{K}}$ suma(-, letcc[Integer](k1.3 + if(x=pred(8),4 * pred(continue(k1,6)), 5 *
suc(continue(cont(not(-); $\Box$),False)))); iszero(-); not(-); $\Box \prec$ 2

$\rightarrow_{\mathcal{K}}$ suma(2,-); iszero(-); not(-); $\Box \succ$ letcc[Integer](k1.3 +
if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); $\Box$),False)))

$\rightarrow_{\mathcal{K}}$ suma(2,-); iszero(-); not(-); $\Box \succ$ 3 +
if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); $\Box$),False))
[k1:= cont(suma(2,-); iszero(-); not(-); $\Box$)]

$\equiv$ suma(2,-); iszero(-); not(-); $\Box \succ$ 3 +
if(x=pred(8),4 * pred(continue(cont(suma(2,-); iszero(-); not(-); $\Box$),6)),
5 * suc(continue(cont(not(-); $\Box$),False))

$\rightarrow_{\mathcal{K}}$ suma(-, if(x=pred(8),4 * pred(continue(cont(suma(2,-); iszero(-); not(-); $\Box$),6)),
5 * suc(continue(cont(not(-); $\Box$),False))); suma(2,-); iszero(-); not(-); $\Box$
$\succ$ 3

$\rightarrow_{\mathcal{K}}$ suma(-, if(x=pred(8),4 * pred(continue(cont(suma(2,-); iszero(-); not(-); $\Box$),6)),
5 * suc(continue(cont(not(-); $\Box$),False))); suma(2,-); iszero(-); not(-); $\Box$
$\prec$ 3

$\rightarrow_{\mathcal{K}}$ suma(3,-); suma(2,-); iszero(-); not(-); $\Box \succ$ if(x=pred(8),4 *
pred(continue(cont(suma(2,-); iszero(-); not(-); $\Box$),6)),
5 * suc(continue(cont(not(-); $\Box$),False)))

$\rightarrow_{\mathcal{K}}$ if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); $\Box$),6)) ,
5 * suc(continue(cont(not(-); $\Box$),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); $\Box \succ$ x=pred(8)

$\rightarrow_{\mathcal{K}}$ =(-,pred(8)); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); $\Box$),6)) ,
5 * suc(continue(cont(not(-); $\Box$),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); $\Box \succ$ x con x=7

$\to_{\mathcal{K}}$ =(-,pred(8)); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); □ ≺ x con x=7
$\to_{\mathcal{K}}$ =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); □ ≻ pred(8)
$\to_{\mathcal{K}}$ pred(-); =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); □ ≻ 8
$\to_{\mathcal{K}}$ pred(-); =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
5 * suc(continue(cont(not(-); □),False)))
5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); □ ≺ 8
$\to_{\mathcal{K}}$ =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
5 * suc(continue(cont(not(-); □),False)))
5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); □ ≻ 7
$\to_{\mathcal{K}}$ =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
5 * suc(continue(cont(not(-); □),False)))
5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); □ ≺ 7
$\to_{\mathcal{K}}$ if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
5 * suc(continue(cont(not(-); □),False)))
5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
suma(2,-); iszero(-); not(-); □ ≺ True
$\to_{\mathcal{K}}$ suma(3,-); suma(2,-); iszero(-); not(-); □ ≻ 4 *
pred(continue(cont(suma(2,-); iszero(-); not(-); □),6))
$\to_{\mathcal{K}}$ *(-, pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)));
suma(3,-); suma(2,-); iszero(-); not(-); □ ≻ 4
$\to_{\mathcal{K}}$ *(-, pred(continue(cont(suma(2,-); iszero(-); not(-); □),6));
suma(3,-); suma(2,-); iszero(-); not(-); □ ≺ 4
$\to_{\mathcal{K}}$ *(4,-); suma(3,-); suma(2,-); iszero(-); not(-); □
≻ pred(continue(cont(suma(2,-); iszero(-); not(-); □),6))
$\to_{\mathcal{K}}$ pred(-); *(4,-); suma(3,-); suma(2,-); iszero(-); not(-); □
≻ continue(cont(suma(2,-); iszero(-); not(-); □),6)
$\to_{\mathcal{K}}$ continue(-,6); pred(-); *(4,-); suma(3,-); suma(2,-); iszero(-); not(-); □
≻ cont(suma(2,-); iszero(-); not(-); □)
$\to_{\mathcal{K}}$ continue(-,6); pred(-); *(4,-); suma(3,-); suma(2,-); iszero(-); not(-); □
≺ cont(suma(2,-); iszero(-); not(-); □)
$\to_{\mathcal{K}}$ continue(cont(suma(2,-); iszero(-); not(-); □),-); pred(-); *(4,-);
suma(3,-); suma(2,-); iszero(-); not(-); □ ≻6
$\to_{\mathcal{K}}$ continue(cont(suma(2,-); iszero(-); not(-); □),-); pred(-); *(4,-);
suma(3,-); suma(2,-); iszero(-); not(-); □ ≺6
$\to_{\mathcal{K}}$ suma(2,-); iszero(-); not(-); □ ≺ 6
$\to_{\mathcal{K}}$ iszero(-); not(-); □ ≺ 8
$\to_{\mathcal{K}}$ not(-); □ ≺ False
$\to_{\mathcal{K}}$ □ ≺ True

Ahora Evaluamos, considerando $x \neq 7$

```
□ ≻ not(letcc[Bool](k2.iszero(2 + letcc[Integer](k1.3 +
    if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(k2,False))))))
→K   not(-); □ ≻ letcc[Bool](k2.iszero(2 + letcc[Integer](k1.3 +
     if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(k2,False)))))
→K   not(-); □ ≻ iszero(2 + letcc[Integer](k1.3 +
     if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(k2,False)))) [k2:= cont(not(-); □)]
≡    not(-); □ ≻ iszero(2 + letcc[Integer](k1.3 +
     if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); □),False))))
→K   iszero(-); not(-); □ ≻ 2 + letcc[Integer](k1.3 +
     if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); □),False)))
→K   suma(-, letcc[Integer](k1.3 + if(x=pred(8),4 * pred(continue(k1,6)), 5 *
     suc(continue(cont(not(-); □),False)))); iszero(-); not(-); □  ≻ 2
→K   suma(-, letcc[Integer](k1.3 + if(x=pred(8),4 * pred(continue(k1,6)), 5 *
     suc(continue(cont(not(-); □),False)))); iszero(-); not(-); □  ≺ 2
→K   suma(2,-); iszero(-); not(-); □ ≻ letcc[Integer](k1.3 +
     if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); □),False)))
→K   suma(2,-); iszero(-); not(-); □ ≻ 3 +
     if(x=pred(8),4 * pred(continue(k1,6)), 5 * suc(continue(cont(not(-); □),False))
     [k1:= cont(suma(2,-); iszero(-); not(-); □)]
≡    suma(2,-); iszero(-); not(-); □ ≻ 3 +
     if(x=pred(8),4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)),
     5 * suc(continue(cont(not(-); □),False))
→K   suma(-, if(x=pred(8),4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)),
     5 * suc(continue(cont(not(-); □),False))); suma(2,-); iszero(-); not(-); □
     ≻ 3
→K   suma(-, if(x=pred(8),4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)),
     5 * suc(continue(cont(not(-); □),False))); suma(2,-); iszero(-); not(-); □
     ≺ 3
→K   suma(3,-); suma(2,-); iszero(-); not(-); □ ≻ if(x=pred(8),4 *
     pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)),
     5 * suc(continue(cont(not(-); □),False)))
→K   if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
     5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
     suma(2,-); iszero(-); not(-); □ ≻ x=pred(8)
→K   =(-,pred(8)); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
     5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
     suma(2,-); iszero(-); not(-); □ ≻ x  con x=7
→K   =(-,pred(8)); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
     5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
     suma(2,-); iszero(-); not(-); □ ≺ x  con x ≠ 7
→K   =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
     5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
     suma(2,-); iszero(-); not(-); □ ≻ pred(8)
```

```
→K  pred(-); =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
    5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
    suma(2,-); iszero(-); not(-); □ ≻ 8
→K  pred(-); =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
    5 * suc(continue(cont(not(-); □),False)))
    5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
    suma(2,-); iszero(-); not(-); □ ≺ 8
→K   =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
    5 * suc(continue(cont(not(-); □),False)))
    5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
    suma(2,-); iszero(-); not(-); □ ≻ 7
→K   =(x,-); if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
    5 * suc(continue(cont(not(-); □),False)))
    5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
    suma(2,-); iszero(-); not(-); □ ≺ 7
→K  if(-, 4 * pred(continue(cont(suma(2,-); iszero(-); not(-); □),6)) ,
    5 * suc(continue(cont(not(-); □),False)))
    5 * suc(continue(cont(not(-); □),False)))); suma(3,-);
    suma(2,-); iszero(-); not(-); □ ≺ False
→K  suma(3,-); suma(2,-); iszero(-); not(-); □ ≻ 5 * suc(continue(cont(not(-); □),False))
→K  *(-,suc(continue(cont(not(-); □),False))); suma(3,-); suma(2,-); iszero(-); not(-); □ ≻ 5
→K  *(-,suc(continue(cont(not(-); □),False))); suma(3,-); suma(2,-); iszero(-); not(-); □ ≺ 5
→K  *(5,); suma(3,-); suma(2,-); iszero(-); not(-); □ ≻ suc(continue(cont(not(-); □),False))
→K  suc(-); *(5,-); suma(3,-); suma(2,-); iszero(-); not(-); □ ≻ continue(cont(not(-); □),False)
→K  continue(-,False); suc(-); *(5,-); suma(3,-); suma(2,-);iszero(-);not(-); □ ≻ cont(not(-);□)
→K  continue(-,False); suc(-); *(5,-); suma(3,-); suma(2,-);iszero(-);not(-); □ ≺ cont(not(-);□)
→K  continue(cont(not(-); □),-); suc(-); *(5,-); suma(3,-);suma(2,-);iszero(-); not(-);□ ≻ False
→K  continue(cont(not(-); □),-); suc(-); *(5,-); suma(3,-);suma(2,-);iszero(-); not(-);□ ≺ False
→K  not(-); □ ≺ False
→K  □ ≺ True
```

3. (1.5pt) Considere la siguiente función `N` que depende de ciertas funciones dadas `f, g, h`.

```
N 0 = 17

N 1 = f (1 + 13)

N 2 = 22 + (2-3) + 2

N 3 = 22 + (f 3) + 37

N 4 = g 22 (f 4)

N 4 = 22 + (f 4) + 33 + (g y)

N x = h (f x) (44 - y) (g y)
```

Defina la versión `cps` de `N`, denotada por `cpsN`. Para esto, puede suponer definidas las versiones `cps` de `f, g, h`, denotadas por `cpsf, cpsg, cpsh`.

Atención: Las operaciones aritméticas deben permanecer sin cambios. Es decir, no se piden las versiones `cps` de `+, -`.

Solución

```
cpsN 0 k = 17

cpsN 1 k = cpsf (1 + 13) (\v -> k v)

cpsN 2 k = k (22 + (2-3) + 2)

cpsN 3 k = cpsf 3 (\v -> k (22 + v + 37))

cpsN 4 k = cpsg 22 (\v1 -> cpsf 4 (\v2 -> k(v1 + v2))

cpsN 5 k = cpsf 4 (\v1 -> cpsg y (\v2 -> k(22 + v1 + 33+ v2)))

cpsN x k = cpsf x (\v1 -> cpsg y (\v2 -> cpsh v1 (44 -y) v2 (\v3 -> k v3)))
```

4. (1.5pt) Considere las siguientes cuatro caracterísiticas para un lenguaje de programación.

   a) Existe un tipo de cadena `String` y se cumple `Int` $\leq$ `String` mediante una conversión implícita que transforma a un entero en una cadena.
      Por ejemplo, `123` a `"123"`.
   b) Existe un tipo cadena `String` que cumple que `String` $\leq$ `Int` mediante una conversión implícita que transforma cadenas a enteros ignorado los caractéres que no sean dígitos, excepto por el caracter inicial `-`.
      Por ejemplo, `"-0aw23r4"` corresponde a `-234`.
   c) Existe un operador binario `+` que denota a ambas la suma de enteros y la concatenación de cadenas.
   d) Existe un operador binario `=` que denota a ambas la igualdad de enteros y de cadenas.

Para cada par de estas caracterísiticas, discuta si se violan o no los principios fundamentales del subtipado. En caso afirmativo escriba un ejemplo de un programa simple que cause un comprotamiento ambigüo o contrainituitivo.

- `String` $\leq$ `Int` e `Int` $\leq$ `String`
  Si se tiene algún operador `*` sobrecargado (como en los demás casos), entonces podría existir amigüedad en la operación.
  ¿Cuál de los dos valores debería recibir el casting?
  En una implementación se podría resolver fijando un elemento como el tipo por omisión para estos caso, pero significaría que la función podría dar diferentes valores en ciertas condiciones dependiendo las desiciones de diseño del lenguaje, independientemente de la definición de la función.
  Por ejemplo, con una expresión `1+"2"`, ¿cuál es el tipo del resultado? Usando únicamente los principios básicos del subtipado, ambos tipos serían correctos, lo que haría que una expresión tenga dos tipos.
- `String` $\leq$ `Int` y `+` sobrecargado
  La cohersión de cadena a entero provoca pérdida de información, lo cuál puede crear casos extraños, si bien los principios del subtipado se mantienen.
  Considere

  ```
  ("1"+"1")+1  = 12
  "1"+("1"+1)  = 3
  ```

  `+` ya no sería asociativa.
- `String` $\leq$ `Int` y `=` sobrecargado
  Como la cohersión de cadena a entero causa pérdida de información, el comportmiento de `=` se vuelve un poco contraintuitivo.
  Considere

  ```
  "1q"  = 1
  1 = "1e"
  "1q"  /= "1e"
  ```

  `=` ya no es transitiva, lo cuál es chocante porque ya no sería una relación de equivalencia.
  A pesar de esto, los principios del subtipado se mantienen.

- `Int` $\leq$ `String` y $+$ sobrecargado

  Con varias aplicaciones del operador con diferentes tipos, el tipo de la operación total dependerá del orden en el que se hagan las operaciones internas.

  Considere

  ```
  (1+2)+"a" = "3a"
  1+(2+"a") = "12a"
  ```

  $+$ ya no sería asociativa.

  Fuera de este inconveniente, todo respecto a los tipos se mantiene en regla.

- `Int` $\leq$ `String` y $=$ sobrecargado

  El único caso donde se haría cohersión sería al tener una cadena y un número. Pero la única cadena a la que el número es igual es a su converisón dígito a dígito, por lo que no se pierde información.

  Por lo que aún cuando se mezclen los argumentos, se mantiene la igualdad.

  No hay ningún aspecto inusual en este caso.

- $+$ y $=$ sobrecargados

  Si suponemos que no se cumple que algún tipo es subtipo del otro, entonces toda aplicación de estos operadores siempre resultarí en operaciones entre cosas del mismo tipo.

  Mezclar tipos en los argumentos resultaría en un error de tipo. Por lo que no hay casos extraños causados por estas mezclas.

  No hay ningún cambio o aspecto contraintuitivo en este caso.

5. (1pt) Usando la definición de números naturales en `Java Peso Pluma` vista en clase.

   a) Agregue un método `pot` y `leq` para las operaciones de potencia y el orden $\leq$.

   Solución:

```
// Clase de naturales definida en las notas
class Nat extends Object {
    Object p;

    Nat (Object n) { super(); this.p = n;}

    Nat suc() { return new Nat(this); }

    Nat pred() { return (Nat) this.p; }

    Nat suma(Nat n) { return this.pred().sum(n.suc());}

    Nat multi(Nat n) { return n.suma(this.pred().multi(n));}

    /** método 'pot'
    * @param Nat n exponente al que se va a elevar la potencia.
    * @return this^n
    */
    Nat pot(Nat n) { return n.flipPot(this) }

    /** método auxiliar para calcular 'pot'
    * @param Nat n base del números del que this es exponente.
    * @return n^this
    */
    Nat flipPot(Nat n) {return n.multi(this.pred().flipPot(n))}

    /** método 'leq'
    * this ≤ n ⟺ this ≯ n ⟺ n ≮ this
    * @param Nat n Número a comparar con this.
    * @return this ≤ n
    */
    Boolean leq(Nat n) {return n.lt(this).not();}
```

```
        /** método auxiliar para calcular 'leq'
        * this < n ⟺ 0 < n − this
        * @param Nat n Número a comparar con this.
        * @return this < n
        */
        Boolean lt(Nat n) {return this.pred().lt(n.pred());}

        // método auxiliar para calcular 'not' en Boolean
        Nat inv() {return new Cero(this);}

    }

    // Clase del Cero definida en las notas
    class Cero extends Nat {
        Cero (Object n) {super(n);}

        Nat pred() {return this;}

        Nat suma(Nat n) {return n;}

        Nat multi(Nat n) {return this;}

        /** calcular el caso base del método auxiliar 'flipPot'
        * @param Nat n número del que this es exponente.
        * @return n^0 = 1 = suc(0) = this.suc()
        */
        Nat flipPot(Nat n) {return this.suc();}

        /** calcular el caso base del método auxiliar 'lt'
        * 0 < n ⟺ bool(n) = True
        * @param Nat n número a comparar
        * @return this < n
        */
        Booelan lt(Nat n) {return new Boolean(n);}

        // método auxiliar para calcular 'not' en Boolean
        Nat inv() {return this.suc();}
    }
```

Donde `not` está definido como

```
class Boolean extends Object {
    ...
    Boolean not() {return new Boolean(this.m.inv());}
}
```

b) Modele la clase `Boolean` para el manejo de valores booleanos

- La clase debe extender de `Object`
- El constructor recibirá un objeto de la clase `Nat` para definir su valor `0` representa `false` y `1` representa `true`.
- Debe tener métodos `true` y `false` que regresen una instancia de `Boolean` según el caso.

Solución:

```
class Boolean extends Object {
    Nat m;

    Boolean (Nat n) { super(); this.m = n;}

    Boolean true() { return new Boolean (new Cero(this).suc());}

    Boolean false() { return new Boolean (new Cero(this));}
```

```
        Boolean not() { return new Boolean(this.m.inv());}
    }
```

6. (2pt) Definina el siguiente lenguaje `MinEAB`

   ```
   e ::= n | true | false | e + e | e < e
   ```

   en `Java Peso Pluma`.

   Hay que seguir los siguientes pasos

   a) Defina una clase `Expr` que incluya los siguientes métodos

      - `isAtom` que devuelve `true` si la expresión no tiene subexpresiones propias.
      - `lsub` que devuelve la subexpresión izquierda de una expresión no atómica.
      - `rsub` que devuelve la subexpresión derecah de una expresión no atómica.
      - `eval` que devuelve el valor de la expresión.

      Esta clase debe ser abstracta en el sentido de que no tiene atributos y por lo tanto sus métodos no tiene cuerpo. Es una interfaz.

      Solución:

      ```
      class Expr extends Object {

          Boolean isAtom() {return error;}

          Expr lsub() {return error;}

          Expr rsub() {return error;}

          Expr eval() {return error;}
      }
      ```

   b) Defina las siguientes clases que implemente a `Expr`

      - `NumExpr` que implemente los métodos para manejar números.
        Solución:

        ```
        class NumExpr extends Expr {
            Nat v;

            NumExpr(Nat w) {super(); this.v = w;}

            Boolean isAtom() {return  (new Boolean(v)).true();}

            Expr eval() {return this;}
        }
        ```

      - `BoolExpr` que implemente los métodos para manejar booleanos.
        Solución:

        ```
        class BoolExpr extends Expr {
            Boolean v;

            BoolExpr(Boolean v) {super(); this.v = v;}

            Boolean isAtom() {return this.v.true();}

            Expr eval() {return this;}
        }
        ```

      - `SumExpr` que implemente los métodos para manejar sumas.
        Solución:

```java
class SumExpr extends Expr {
    Expr i;
    Expr d;

    SumExpr(Expr e1, Expr e2) {super(); this.i = e1; this.d = e2}

    Boolean isAtom() {return (new Booelan(new Cero(this))).false();}

    Expr lsub() {return this.i;}

    Expr rsub() {return this.d;}

    Expr eval() {return new NumExpr(this.i.eval().v.suma(this.d.eval().v));}
}
```

- `LTExpr` que implemente los métodos para manejar comparaciones de orden.

  Solución:

```java
class LTExpr extends Expr {
    Expr i;
    Expr d;

    LTExpr(Expr e1, Expr e2) {super(); this.i = e1; this.d = e2}

    Boolean isAtom() {return (new Boolean(new Cero(this))).false();}

    Expr lsub() {return this.i;}

    Expr rsub() {return this.d;}

    Expr eval() {return new BoolExpr(this.i.eval().v.lt(this.d.eval().v));}
}
```

c) Dé ejemplos de instancias de cada una de las clases anteriores.

```java
// Cero
Nat cero = new Cero(this);

// NumExpr
NumExpr ceroE = new NumExpr(cero);

// BoolExpr
BoolExpr falseE = new BoolExpr(new Boolean(cero));

// SumExpr
SumExpr sumCE = new SumExpr(ceroE, ceroE);

// LTExpr
LTExpr lC = new LTExpr(ceroE, ceroE);
```

d) Extienda `MinEAB` con las expresiones `-e, iszero e`.
Con esta nueva definición, cree las siguientes clases

- `NegExpr`

  Solución:

  Como ahora hay números negativos, hay que definir la clase de enteros.

```java
// definición de enteros usando naturales
// parejas de aturales cuya diferencia indica el número a
// representar.
// n ∈ ℕ
// n ↦ (n,0)
// −n ↦ (0,n)
class Int extends Object {
```

```
    Nat i;
    Nat d;

    // $\mathbb{Z} \cong \mathbb{N}^2$
    Int(Nat i, Nat d) {super(); this.i = i; this.d = d;}

    // $(x, y) +_{\mathbb{Z}} (z, w) = (x +_{\mathbb{N}} y, z +_{\mathbb{N}} w)$
    Int suma(Int n) {return new Int(this.i.suma(n.i), this.d.suma(n.d));}

    // $(x, y) \leq_{\mathbb{Z}} (z, w) \iff xz \leq_{\mathbb{N}} yz$
    Boolean lt(Int n) {return this.i.suma(n.d).lt(n.i.suma(this.d));}

    // $-(a, b) = (b, a)$
    Int neg() {return new Int(this.d, this.i);}
}
```

Hay que modificar la clase de `NumExpr` para manejar enteros

```
// modificaciones para manejar enteros
class NumExpr extends Expr {
    Int v;

    NumExpr(Int v) {super(); this.v = v;}
    ...
}
```

Y con estas nuevas modificaciones se define `NegExpr`

```
// manejar expresiones con negativos
class NegExpr extends Expr {
    Expr e;

    NegExpr(Expr e) {this.e = e;}

    Boolean isAtom() {return (new Booelan(new Nat(this))).false();}

    Expr eval() {return this.e.eval().neg();}
}
```

- `IsZero`

  Para evaluar esto, se modifica la clase `Nat` agregando los siguientes método.

```
// algunas modificaciones para Booleanos
class Boolean extends Object {
    ...
    /**
    * $p \lor q = False \iff p = q = False = Bool(0) = Bool(0 + 0)$
    */
    Boolean or(Boolean p) {return new Boolean(this.m.suma(p.m));}

    /**
    * $p \land q = \neg(\neg p \lor \neg q)$
    */
    Boolean and(Boolean p) {return this.not().or(p.not()).not();}
}

// modificaciones para Nat
class Nat extends Object {
    ...
    /**
    * $a = b \iff a \leq b \land b \leq a$
    */
    Nat eq(Nat n) {return this.leq(n).and(n.leq(this));}
    ...
```

```
}

// modificacioes para Int
class Int extends Object {
    ...
    Booelan isZero(Int i) {return this.i.eq(this.d);}
    ...
}
```

Y se usa este método para

```
class IsZExpr extends Expr {
    Expr e;

    IsZExpr(Expr e) {super(); this.e = e;}

    Boolean isAtom() {return (new Booelan(new Nat(this))).false();}

    Expr eval() {return new BoolExpr(e.eval().v.isZero());}
}
```

e) Dé ejemplos de instancias de estas dos clases.

```
// cero
Int ceroI = new Int(new Cero(this), new Cero(this));

// NegExpr
NegExpr ceroE = new NegExpr(ceroI);

// IsZero
IsZero isZE = new IsZExpr(ceroI);
```

f) ¿Cómo se modifican las subclases de `Expr` definidas en puntos anteriores?

Para manejar `NegExpr`, definió la clase `Int` y se remplazó `Nat` en `NumExpr` por `Int`.

Para definir `IsZero`, sólo se modificaron las clases de `Boolean`, `Nat` y `Int` agregando los métodos necesarios para usar `isZero` en `Int`.

Puede suponer definida la clase `Value` (escencialmente `Nat + Bool`) cuyas instancias sean los valores del lenguaje.

Además de otras clases primitivas con los métodos que requiera.

También se puede usar la constantee de error en cualquier método.

7. **Extra (hasta 2pt): Privacidad en** `Java Peso Pluma`

`Java` proporciona mecanismos para controlar el accesso. Un método o atributo de una clase pueden declararse como *público, protegido o privado*. En `Java Peso Pluma` es posible agregar este mecanismo como sigue

```
C ::= class C extends C {p⃗ C⃗ f⃗ ;K M⃗ } // declaración de clases
M ::= p C m (C⃗  x⃗ ) {return e;} // declaración de métodos
p ::= public | protected | private // modificacor de privacidad
```

EL significado intuitivo de los modificadores de privacidad es el siguiente:

- Si un método o atributo de una clase `C` se declara `public`, entonces se permite el acceso desde cualquier lugar.
- Si un método o atributo de una clase `C` se declara `protected`, entonces se permite el acceso únicamente desde métodos de `C` y subclases de `C`.
- Si un método o atributo de una clase `C` se declara `private`, entonces se permite el acceso únicamente desde métodos de `C`.

Extienda la semática estática para filtrar programas que contengan violaciones de privacidad de acuerdo a las reglas dadas arriba. Especifique claramente cuales reglas de tipado originales se eliminan o se sustituyen por nuevas y cuáles se mantienen.

Sugerencia: Cuando se verifique si un método está bien formado en una clase `C`, se debe verificar si la expresión en el cuerpo del método no se refiere a métodos o atributos en otra clase `D` que sean privados o protegidos si $\mathtt{C} \not\leq \mathtt{D}$. Podría necesitarse el paso de algo más en el contexto Γ en el juicio de tipado para expresiones.

Solución:

Primero, hay que restringir el acceso de los atributos, que está dado orignalmente por la regla `T-FLD`

Los atributos públicos no cambian

$$\frac{\Gamma, Caller(this) : Call(E) \vdash e : C \qquad fields(C) = \vec{\mathtt{p}}\ \vec{\mathtt{C}}\ \vec{\mathtt{f}} \qquad p_i = public}{\Gamma, Caller(this) : Call(E) \vdash e.f_i : C_i} \text{ T-FLD-PUB}$$

Los atributos protegidos requieren que la clase que llama sea subtipo de la clase donde están los atributos.

$$\frac{\Gamma, Caller(this) : Call(E) \vdash e : C \qquad fields(C) = \vec{\mathtt{p}}\ \vec{\mathtt{C}}\ \vec{\mathtt{f}} \qquad p_i = protected \qquad E \leq C}{\Gamma, Caller(this) : Call(E) \vdash e.f_i : C_i} \text{ T-FLD-PROC}$$

Los atributos privados solo se puede acceder desde la clase misma.

$$\frac{\Gamma, Caller(this) : Call(E) \vdash e : C \qquad fields(C) = \vec{\mathtt{p}}\ \vec{\mathtt{C}}\ \vec{\mathtt{f}} \qquad p_i = private \qquad E = C}{\Gamma, Caller(this) : Call(E) \vdash e.f_i : C_i} \text{ T-FLD-PRIV}$$

Se requieren modificaciones análogas a la regla `T-INV` para la invocación de métodos.

La invocación de métodos públicos no requiere verificación

$$\frac{\Gamma, Caller(this) : Call(E) \vdash e_0 : C_0 \qquad \Gamma \vdash \vec{e_0} : \vec{C_0} \qquad mtype(m, C_0) = p\ \vec{D} \to C \qquad \vec{C} \leq \vec{D} \qquad p = public}{\Gamma, Caller(this) : Call(E) \vdash e_0.m(\vec{e}) : C} \text{ T-INV-PUB}$$

Para los métodos protegidos, la clase que invoca debe ser subtipo

$$\frac{\Gamma, Caller(this) : Call(E) \vdash e_0 : C_0 \qquad \Gamma \vdash \vec{e_0} : \vec{C_0} \qquad mtype(m, C_0) = p\ \vec{D} \to C \qquad \vec{C} \leq \vec{D} \qquad E \leq C_0 \qquad p = protected}{\Gamma, Caller(this) : Call(E) \vdash e_0.m(\vec{e}) : C} \text{ T-INV-P}$$

Para métodos privados, la clase que invoca y la clase que tiene los métodos deben ser la misma

$$\frac{\Gamma, Caller(this) : Call(E) \vdash e_0 : C_0 \qquad \Gamma \vdash \vec{e_0} : \vec{C_0} \qquad mtype(m, C_0) = p\ \vec{D} \to C \qquad \vec{C} \leq \vec{D} \qquad E = C_0 \qquad p = private}{\Gamma, Caller(this) : E \vdash e_0.m(\vec{e}) : C} \text{ T-INV-PR}$$

Además, es necesario agregar la clase que invoca al contexto de las expresiones que forma el cuerpo de los métodos durante la revisión de la buena formación de métodos.

$$\frac{\begin{array}{c} T(C) = class\ C\ extends\ D\{...\} \\ mtype(m, D) = p\ \vec{C} \to C_0 \\ \vec{x} : \vec{C}, this : C, Caller(this) : Call(C) \vdash e_0 : C_0' \\ C_0' \leq C_0 \end{array}}{\text{p } C_0 \text{ m \{return } e_0 \text{ ;\} ok in C}}$$