Problema I: *Haga el juicio de tipo para la función fibonacci y el predicado empty?*

Función Fibonacci:

```
{rec {fib {fun {n}
        {if {or {zero? n} { zero? {- n 1}}}} 1
        { + {fib {- n 1}} {fib { - n 2}}}}}}
```

Paso 1:

Γ ⊢ [fib ← number] ⊢ {fun {n}
{if {or {zero? n} { zero? {- n 1}}}} 1
{ + {fib {- n 1}} {fib { - n 2}}}} : number

Paso 2:

Γ ⊢ {fun {n}
{if {or {zero? n} { zero? {- n 1}}}}1
{ + {fib {- n 1}} {fib { - n 2}}}} : number

Paso 3:

Γ ⊢ {fun {n}
{if {or {zero? n} { zero? {- n 1}}}}1
{ + {fib {- n 1}} {fib { - n 2}}}} :(number → number)

Paso 4:

Γ ⊢ [n ← number] ⊢
{if {or {zero? n} { zero? {- n 1}}}}1
{ + {fib {- n 1}} {fib { - n 2}}}

Paso 5:

Γ ⊢ or: boolean ⊢ {zero? n} { zero? {- n 1}
Γ ⊢ { + {fib {- n 1}} {fib { - n 2}}}

Paso 6:

Γ ⊢ {zero? n}: boolean Γ ⊢ { zero? {- n 1}}: boolean Γ ⊢ 1: number
Γ ⊢ { + {fib {- n 1}} {fib { - n 2}}}: number

Paso 7:

Γ ⊢ {zero? n}: boolean Γ ⊢ n: number Γ ⊢ { zero? {- n 1}}: boolean
Γ ⊢ {- n 1}: number Γ ⊢ {fib {- n 1}}: number Γ ⊢{fib {- n 2}}: number

Paso 8:

Γ ⊢ n:number Γ ⊢ 1 :number Γ ⊢n:number Γ ⊢1 :number Γ ⊢ n: number Γ ⊢ 2: number

Empty?:
(empty? e)

Paso 1:
Γ ⊢ (empty? e): boolean

Paso 2:
Γ ⊢ [empty? ← e] : boolean

Paso 3:
Γ ⊢ e: list;

Problema II: *Considera el siguiente programa:*
(+ 1 (first (cons true empty)))
Este programa tiene un error de tipos.
Genera restricciones para este programa. Aísla el conjunto más pequeño de estas restricciones tal que, resultas juntas, identifiquen el error de tipos.
Siéntete libre de etiquetar las sub-expresiones del programa con superíndices para usarlos cuando escribas y resuelvas tus restricciones.

①(+ ②1 ③(first ④(cons ⑤true ⑥empty)))

[| ① |] = [| (+ 1 (first (cons true empty))) |]
= [|+ ② ③ |] = number
y [| ② |] = number
[| ③ |] = number

[| ② |] = [| 1 |] = number

[| ③ |] = [| (first (cons true empty)) |]
= [| ③ |] = number
y [| ④ |] = list

[| ④ |] = [| (cons true empty) |]
= [| ④ |] = list
y [| ⑤ |] = number
[| ⑥ |] = list

[| ⑤ |] = [| true |] = boolean

[| ⑥ |] = [| empty |] = list

Esto es un error semántico, ya que no se puede sumar números con booleanos, en este caso la sub-expresión ③ debe ser de tipo number por la suma y sin en cambio al obtener el valor de esa subexpresión nos devuelve ⑤ con un booleano, por lo que al momento de la ejecución éste nos devolverá un error.

Restricciones:
Dado que es una suma la función

1. Sean e1 y e2 expresiones
   [| + e1 e2 |] = number
   [| e1 |] = number
   [| e2 |] = number

2. Sea e expresión
   [| e |] = number

3. [| first e |] = number
   [| e |] = list

4. [| cons e1 e2 |] = list
   [| e1 |] = number
   [| e2 |] = list

5. [| true |] = boolean

6. [| empty |] = list

Problema III: *Considera la siguiente expresión con tipos:*

```
{fun {f : C1 } : C2
  {fun {x : C3 } : C4
    {fun {y : C5 } : C6
      {cons x {f {f y}}}}}}
```

Dejamos los tipos sin especificar (Cn) para que sean llenados por el proceso de inferencia de tipos. Deriva restricciones de tipos para el programa anterior. Luego resuelve estas restricciones. A partir de estas soluciones, rellena los valores de las Cn. Asegúrate de mostrar todos los pasos especificados por los algoritmos (i.e., escribir la respuesta basándose en la intuición o el conocimiento es insuficiente). Deberás usar variables de tipo cuando sea necesario. Para no escribir tanto, puedes etiquetar cada expresión con una variable de tipos apropiada, y presentar el resto del algoritmo en términos solamente de estas variables de tipos.

Como se tienen que inferir los tipos de las Cn, comenzamos el proceso con el cons:
① {cons ②x ③{f ④{f ⑤y}}}

[| ① |] = [| {cons ②x ③{f ④{f ⑤y}}} |]
= [| ① |] = list
[| ② |] = number
[| ③ |] = list

[| ② |] = [| x |] = number

[| ③ |] = [| {f {f y}} |]
= [| y |] → [| {f y} |] = list

[| ④ |] = [| {f y} |]
= [| y |] → [| y |]

[| ⑤ |] = [| y |] = number

De acuerdo a estas restricciones podemos decir que:

```
{fun {f : number } : list
  {fun {x : number } : list
    {fun {y : number } : list
      {cons x {f {f y}}}}}}
```

Problema IV: *Considera los juicios de tipos discutidos en clase para un lenguaje glotón (en el capítulo de Juicios de Tipos del libro de Shriram). Considera ahora la versión perezosa del lenguaje. Pon especial atención a las reglas de tipado para:*

- *definición de funciones*
- *aplicación de funciones*

Para cada una de estas, si crees que la regla original no cambia, explica por qué no (Si crees que ninguna de las dos cambia, puedes responder las dos partes juntas). Si crees que debe cambiar algún otro juicio de tipos, mencionalo también.

Al realizar un juicio de tipos de un lenguaje perezoso, tanto en definición de funciones como su aplicación, no cambiará la forma de juzgar de qué tipo son, ya que ocupará la misma sintaxis; tendrá en cuenta las mismas expresiones para decidir si dicha expresión es de algún tipo ya definido como number o tipo T_n.

Los únicos cambios que ocurrirán serán la forma en la que el árbol se deriva, es decir, los llamados *type judgment tree*. Esto es debido a que en lugar de juzgar un tipo y éste ya no aparezca una vez evaluado como lo hace en el lenguaje glotón, en el caso de evaluación perezosa, él cargará con la expresión completa en el *type environment* tantas veces como sea necesario hasta que se lleve a cabo el juicio de tipos.

Entonces podemos concluir que cuando se realiza un juicio de tipo únicamente se van a inferir los tipos de la función (ya sea para aplicación o definición) y se obtendrá con la misma sintaxis que lo hace con la evaluación glotona.

Ya sea glotón o perezoso, no cambia nada en el juicio de tipo, sino la forma en que repetidas veces aparecerá el ambiente de tipos en el árbol de juicio de tipos.

Problema V: *¿Cuáles son las ventajas y desventajas de tener polimorfismo explícito e implícito en los lenguajes de programación?*

Ventajas del polimorfismo explícito:

- Es más claro obtener información sobre el comportamiento que se esté llevando a cabo en el programa porque conocemos los tipos, tanto los que recibe como los que regresa.
- Ayuda a la documentación de un programa.
- El compilador se ejecuta más rápido, puesto que conoce los tipos con los que se debe tratar, entonces se detectarán los errores en tiempo de compilación, es decir, antes de ejecutarlos.
- Se asegura el comportamiento del programa ya que tiene un tipo especificado.
- Es el único mecanismo que el diseñador del lenguaje da para la introducción de tipos parametrizados, los cuales ayudan en la reutilización de código .
- El lenguaje incluye algunas maquinarias adicionales así que el programador no tiene que escribir todos los tipos cada vez.
- Cuando se es tipado y se detecta algún error, se ayuda a reducir el tiempo de depuración de un programa.

Desventajas del polimorfismo explícito:

- Por cada tipo que se tenga en el lenguaje, se deberá implementar las construcciones necesarias para ese mismo tipo, lo cual se volverá un fastidio; un ejemplo sería construir una lista tanto para números, símbolos, cadenas, etc.
- Se ve difícil de manejar ya que por cada expresión que se declare durante el programa se debe marcar los tipos que recibirán y devolverán.
- Se requiere hacer cast para cambiar su tipo.

Ventajas del polimorfismo implícito:

- Ya que no se especifican los tipos, se tendrá un tipo general, lo cual volverá los programas más cortos, sin tener expresiones con muchas variantes; un ejemplo serían el tipo genérico que se utiliza en Java.
- No se requieren declarar explícitamente los tipos de cada expresión que se cree.
- Se puede cambiar su tipo sin necesidad de hacer un cast.
- Se podría usar un valor varias veces en el mismo contexto de tipos, y la misma expresión en otro lugar varias veces en un contexto de tipos diferente, pero no combina las dos copias del código a través de una unión.

Desventajas del polimorfismo implícito:

- Sin saber qué se recibirá como un valor (el cual no podemos saber hasta el tiempo de ejecución), no podremos estar seguros que ellos de hecho son polimórficos.

- Se deberá decidir el tiempo de chequeo de tipos o no tratarlos como solo un identificador polimorfista, así que estamos forzados a tratarlos monomórficamente
- No se sabe el comportamiento que tendrá el programa ya que puede recibir algo de cualquier tipo menos del esperado y ante esto el programa no sabrá cómo responder, por lo que nos regresará un error.
- No se detectan los errores de tipo hasta el tiempo de ejecución.
- Todo el comportamiento se verá afectado por las entradas que reciba en la ejecución.
- El checador de tipos no detectará problemas en tiempo de compilación ya que sintácticamente se encontrará correcto el programa.

Problema VI Da las ventajas y desventajas de tener lenguajes de dominio específico (DSL) y de propósito general. También da al menos tres ejemplos de lenguajes DSL, cada ejemplo debe indicar el propósito del DSL y un ejemplo documentando su uso.

Ventajas de un lenguaje de dominio específico:

- Conveniencia de notación, usualmente proporcionando una sintaxis que es cercana a establecer normas en el dominio pero lejos de la sintaxis de GPLs.
- Mucha mejor el rendimiento puesto que la implementación de los DSL saben algo sobre el dominio.
- Una semántica no estandarizada: por ejemplos, cuando ni la evaluación perezosa ni la glotona es apropiada.
- Garantiza el rendimiento al restringir su lenguaje de manera significativa. Un ejemplo de esto sería el cálculo lambda

Desventajas de un lenguaje de dominio específico:

- Se tiene que definir su dominio, lo cual no es sencillo ya que si tu audiencia no entiende cuál es, no le pondrá atención a tu lenguaje.
- Se requiere tener un amplio conocimiento del lenguaje para poder manejarlo y resolver los problemas de manera correcta.
- Por las mismas restricciones que realiza el lenguaje, no es tan sencillo hacer una combinación con otro lenguaje

Ventajas de un lenguaje de dominio general:

- En los lenguajes envolventes trabajan muy bien cuando son usados para tareas simples.
- En los lenguajes están incrustados en una aplicación, y exponen parte de la funcionalidad de la aplicación para un programador quien quiera personalizarlo.

Desventajas de un lenguaje de dominio general:

- Los lenguajes envolventes deben proveer suficientes capacidades de abstracción para expresar una amplia variedad de controles, en los cuales traen estructura de datos a través de la puerta trasera.
- El resultado de la red es que tales lenguajes muy seguido son simples lenguajes pero crecen en una manera inmanejable.
- Los lenguajes incrustados dentro de otro lenguaje tienen problemas sintácticos.

- Pueden los lenguajes incrustados acceder a valores desde el lenguajes que está encerrado.
- Muy seguido, los DSL son capaces de hacer garantías del rendimiento solo porque se restringe su lenguaje de alguna manera significativa

Ejemplos de lenguajes DSL:

- ***Fortran:*** Soporte para traducción de fórmulas
- ***Algol:*** Es un lenguaje algorítmico
- ***LISP:*** Es la abreviación de “lista de procesamiento”

Ejemplo del uso de FORTRAN:

En este ejemplo se realiza la suma de dos matrices A y B, con tamaño nxm, y el resultado de la misma es guardado en la matriz R

```
SUBROUTINE MTXADD(A,B,R,N,M,NMAX)
 * ----------------------------------------
IMPLICIT NONE
INTEGER N,M,NMAX
REAL A(NMAX,*)
REAL B(NMAX,*)
REAL R(NMAX,*)
* ----------------------------------------
INTEGER I,J
* ----------------------------------------
DO I=1,N
        DO J=1,M
                R(I,J) = A(I,J)+B(I,J)
        END DO
END DO RETURN
```