Tarea 2
Inferenciador de tipos

Para la resolución de la tarea, recuerde que

- La tarea es individual o de a dos. En caso de hacerla de a dos, debe entregarla sólo uno de los miembros del grupo, y recuerde incluir los nombres de ambos miembros en el archivo `base.rkt`. (Una vez entregada la tarea, no se permiten modificaciones en la composición del grupo.)
- En el material docente (`Clase 01.pdf`, diapositivas 7 y 8) se publicó la política de colaboración, reglamentando como está autorizado colaborar con compañer@s de otro grupo.
- Para resolver los distintos ejercicios, se permite definir funciones auxiliares. Éstas deben ir acompañada de su firma y una breve descripción coloquial.
- Todas las funciones presentes en el archivo `base.rkt` (auxiliares o ya incluidas) deben ir acompañadas de un conjunto "significativo" de tests en el archivo `test.rkt`.
- Las funciones que no lleven firma, descripción coloquial o tests no serán consideradas para la evaluación.
- De la misma forma, las funciones que no cumplan con las interfaces definidas en la tarea no serán consideradas para la evaluación.
- Los mensajes que se deban imprimir hacia el usuario (e.g. mensajes de error), deben ser idénticos a los especificados en enunciado de la tarea.
- La entrega vía U-Cursos debe constar de dos archivos: uno donde se encuentre el código fuente de la tarea (`base.rkt`) y otro con todos los tests que se usaron durante el desarrollo de ésta (`test.rkt`).
- Hay un plazo de dos semanas para la entrega. Se permiten hasta tres días de retraso, con una penalización de medio punto por día.

En esta tarea vamos a equipar el lenguaje visto en clases (y provisto en el archivo base.rkt) con un sistema de *inferencia de tipos*. Esto significa que vamos a checkear que un programa esté bien tipado sin requerir especificaciones de tipos.

En particular, el sistema de tipos que vamos a desarrollar usa variables de tipos para denotar cada tipo desconocido y genera, al analizar un programa, una lista de *constraints* sobre estas variables. Por ejemplo, para la expresión `(fun (x) (+ x 1))`, el inferenciador hace (muy informalmente) lo siguiente:

- Genera una nueva variable de tipo `T1` para denotar el tipo desconocido de `x`.

- Genera la constraint `T1=TNum` para reflejar que esta es la única forma con que el cuerpo de la función pueda estar bien tipado.
- Obtiene que la expresión tiene el tipo `T1->TNum` con la constraint `T1=TNum` generada anteriormente.
- Aplica la substitución de `T1` por `TNum`.
- Retorna al usuario que la expresión tiene el tipo `(TNum->TNum)`.

En las siguientes secciones se muestra paso a paso lo que usted tiene que hacer para poder implementar el inferenciador de tipos. Para ello considere las definiciones adicionales provistas en el archivo *base.rkt*.

Definiciones

Para representar el tipo de una expresión se usará el siguiente tipo de dato

```
(deftype Type
  (TNum)
  (TFun Type Type)
  (TVar Symbol))
```

en donde

- `(TNum)` es el tipo numérico
- `(TFun Type Type)` es el tipo de una función `(Type->Type)`
- `(TVar Symbol)` es una variable de tipo con un `Symbol` único. Cada vez que necesite definir una nueva variable de tipo, use la función `get-id` provista en *base.rkt*.

Finalmente, para representar una constraint `T1=T2` usamos el tipo

```
(deftype Constraint
  (Cnst T1 T2))
```

Ejercicio 1 (TypeOf) — 20 Pt

Para esta sección se define el siguiente *ambiente de tipos*, que asocia un identificador con un tipo (de forma similar a en clase, en donde se definió un ambiente que asocia un identificador con un valor).

```
(deftype TEnv
  (mtEnv)
  (anEnv id Type env))
```

(a) [4 Pt] Defina las funciones

- `emptyT-env :: TEnv`, que construye un ambiente de tipos vacío;
- `extendT-env :: Sym x Type x TEnv -> TEnv`, que extiende un ambiente asociando un tipo a un identificador dado;
- `lookupT-env :: Sym x TEnv -> Type`, que dado un identificador y un ambiente de tipos, retorna el tipo asociado al identificador.

(b) [16 Pt] Defina la función `typeof :: Expr x TEnv -> (Type, List[Constraint])` que dada una expresión y un ambiente de tipos, retorna el tipo de la expresión con la lista de constraints que debe ser solucionable para que el programa sea válido a nivel de tipos. La función debe reportar errores sólo en caso de identificadores libres.

Para obtener el tipo de una expresión siga las siguientes reglas:

- El tipo de un número es `(TNum)`.
- El tipo de la suma (y resta) es `(TNum)`.
- El tipo de un identificador es el tipo al cual esta asociado en el ambiente. Esto puede lanzar un error de identificador libre en caso de que el identificador no este asociado en el ambiente.
- El tipo de una función es de la forma `(TFun A B)` donde `A` es una variable de tipo fresca, es decir, de la forma `(TVar n)` en donde `n` es un nuevo id, y `B` es el tipo del cuerpo de la función, calculado usando el ambiente extendido que asocia el tipo `A` al parámetro formal de la función.
- El tipo de una aplicación es también una variable de tipo fresca.
- El tipo del `if0` es el tipo de la rama `tb`.

Para obtener la lista de constraints siga las siguientes reglas:

C(n)	=	{}
C(x)	=	{}
C({+ a1 a2})	=	C(a1);C(a2);{T1 = TNum};{T2 = TNum} donde T1 es el tipo de a1 y T2 es el tipo de a2 (calculados recursivamente)
C({- a1 a2})	=	C(a1);C(a2);{T1 = TNum};{T2 = TNum} donde T1 es el tipo de a1 y T2 es el tipo de a2
C({if0 e tb fb})	=	C(e);C(tb);C(fb);{Te = TNum};{T1 = T2} donde Te es el tipo de e, T1 es el tipo de tb y T2 es el tipo de tf
C({fun param body})	=	C(body)
C({app fun arg})	=	C(fun);C(arg);{T1 = T2 -> Tx} donde T1 es el tipo de fun, T2 el tipo de arg y Tx la variable de tipo fresca que se asocia a la aplicacion (siguiendo la regla 5 de arriba).

Observe que si bien por fines didácticos se presentó de manera independiente cómo la función `typeof :: Expr x TEnv -> (Type, List[Constraint])` calcula el tipo y cómo calcula la lista de constraints, es posible (y recomendable) no separar la definición en dos funciones, sino tratarlas simultáneamente. (La definición así generada es más limpia).

Ejemplos

```
>(typeof  (num 3) (emptyT-env))
```

```
( list  (TNum))

>(typeof  (add (num 10) (num 3)) (emptyT-env))
( list  (TNum) (Cnst (TNum) (TNum)) (Cnst (TNum) (TNum)))

>(typeof  (if0  (num 2) (num 5) (num 3)) (emptyT-env))
( list  (TNum) (Cnst (TNum) (TNum)) (Cnst (TNum) (TNum)))

>(typeof  (id 'x) (extendT-env 'x (TNum) (emptyT-env)))
( list  (TNum))

>(typeof  (add (num 10) (id 'x)) (extendT-env 'x (TVar 1) (emptyT-env)))
( list  (TNum) (Cnst (TNum) (TNum)) (Cnst (TVar 1) (TNum)))

>(typeof  (app (fun 'x (id  'x)) (num 3)) (emptyT-env))
( list  (TVar 1) (Cnst (TFun (TVar 2) (TVar 2)) (TFun (TNum) (TVar 1))))

>(typeof  (fun 'x (add (id  'x) (num 1))) (emptyT-env))
( list  (TFun (TVar 1) (TNum)) (Cnst (TVar 1) (TNum)) (Cnst (TNum) (
    TNum)))

>(typeof  (fun 'f (fun 'x (app (id  'f) (app (id  'f) (id  'x))))) (emptyT-env))
( list  (TFun (TVar 1) (TFun (TVar 2) (TVar 3))) (Cnst (TVar 1) (TFun (TVar
    2) (TVar 4))) (Cnst (TVar 1) (TFun (TVar 4) (TVar 3))))
```

Nota: Note que en estos ejemplos las llamadas consecutivas de `typeof` retornan los indices siempre partiendo de `1`. No es necesario que usted implemente este comportamiento, pero si lo desea se le provee la función `reset` para reiniciar la numeración de `get-id`.

Ejercicio 2 (Unify) — 20 Pt

Para saber si la lista de constraints generada por `typeof` admite una solución, y en tal caso, obtener la "mejor" solución, debemos usar un proceso llamado unificación. Por ejemplo, luego de aplicar el proceso de unificación a las listas de constraints de los últimos tres ejemplos, obtenemos:

```
>(unify  ( list  (Cnst (TFun (TVar 2) (TVar 2)) (TFun (TNum) (TVar 1)))))
( list  (Cnst (TVar 1) (TNum)) (Cnst (TVar 2) (TNum)))

>(unify  ( list  (Cnst (TVar 1) (TNum)) (Cnst (TNum) (TNum))))
( list  (Cnst (TVar 1) (TNum)))

>(unify  ( list  (Cnst (TVar 1) (TFun (TVar 2) (TVar 4))) (Cnst (TVar 1) (TFun (
    TVar 4) (TVar 3)))))
( list  (Cnst (TVar 4) (TVar 3)) (Cnst (TVar 2) (TVar 4)) (Cnst (TVar 1) (TFun (
    TVar 2) (TVar 4))))
```

Para poder definir el proceso de unificación vamos a hacer uso de varias funciones auxiliares:

(a) [4 Pt] Defina la función `substitute:: TVAR x Type x List[Constraint]-> List[Constraint]` que reemplaza una variable de tipo por otro tipo dado en una lista de constraints. (De aquí en adelante usaremos `TVAR` para representar el tipo de las variables de tipo, es decir, los tipos formadas sólo mediante el constructor `TVar`.)

(b) [4 Pt] Defina la función `occurs-in? :: TVAR x Type -> Bool` que verifica si una variable de tipo ocurre como subexpresión de otro tipo dado.

Ahora ya tenemos todos los prerequisitos para definir el proceso de unificación.

(c) [12 Pt] Defina la función `unify :: List[Constraint] -> List[Constraint]` que dada una lista de constraints retorna la lista *unificada* de constraints. El algoritmo de unificación es el siguiente[1]:

```
unify(C) = if C es vacio
              Retornar lista vacia.
           else
              Sea C = {T1 = T2} + C' //C tiene una constraint y un resto
              if T1 == T2
                 unify(C')  //Seguir con el resto de C
              else if T1 es una variables de tipo y T1 no ocurre dentro de T2
                 unify(substitute(T1 T2 C')) + {T1 = T2} //Se asocia T1 a T2
              else if T2 es una variables de tipo y T2 no ocurre dentro de T1
                 unify(substitute(T2 T1 C')) + {T2 = T1} //Se asocia T2 a T1
              else if T1 es (TFun T1a T1r) y T2 es (TFun T2a T2r)
                 unify(C' + {T1a = T2a} + {T1r = T2r})
              else
                 Error "Exception: Type error: cannot unify T1 with T2"
```

Ejercicio 3 (RunType) — 20 Pt

El último paso que debemos seguir para inferir el tipo "final" de una expresión es usar la lista de constraints unificada para recuperar el tipo final de cada variable de tipo. Por ejemplo, considere nuevamente la expresión

```
(fun 'f (fun 'x (app (id 'f) (app (id 'f) (id 'x)))))
```

que define la función *apply-twice*. Su tipo devuelto por `typoeof` es

```
(TFun (TVar 1) (TFun (TVar 2) (TVar 3)))
```

y su lista de constraints unificada es

```
( list  (Cnst (TVar 4) (TVar 3)) (Cnst (TVar 2) (TVar 4)) (Cnst (TVar 1) (TFun (
    TVar 2) (TVar 4))))
```

La idea es que al encontrar la variable de tipo "(TVar 1)" en el tipo de la expresión, el inferenciador recorra repetidamente la lista hasta concluir que debe reemplazarla por

[1] Los detalles del algoritmo se encuentran en el *Types and Programming Languages*, B. Pierce.

```
(TFun (TVar 3) (TVar 3))
```

La secuencia de reemplazos completa que lleva a cabo es

```
(TVar 1) -> (TFun (TVar 2) (TVar 4)) -> (TFun (TVar 4) (TVar 4)) -> (TFun (TVar
    3) (TVar 4)) -> (TFun (TVar 3) (TVar 3))
```

y se obtiene reemplazando el tipo que aparece a la izquierda de una constraint por el tipo que aparece a la derecha. De manera similar, debe concluir el reemplazo de la variable de tipo `(TVar 2)` por `(TVar 3)`, mientras que `(TVar 3)`, la última variable de tipo que aparece en el tipo de la expresión, debe permanecer igual ya que no aparece a la izquierda de ninguna constraint. En conclusión, el inferenciador anunciará que el tipo de la expresión es

```
((TFun (TFun (TVar 3) (TVar 3)) (TFun (TVar 3) (TVar 3)))
```

(a) [20 Pt] Defina la función `runType :: S-Expr -> Type` que dada una s-expresión, retorne su tipo (o arroje un error) siguiendo el mecanismo recién descrito.

Ejemplos

```
> (runType '(fun (x) (+ x 1)))
(TFun (TNum) (TNum))

> (runType '(fun (x) x))
(TFun (TVar 1) (TVar 1))

> (runType '(fun (x) 3))
(TFun (TVar 1) (TNum))

> (runType 'x)
Exception: free  identifier  x

> (runType '((fun (x) (+ x 1)) (fun (x) x)))
Exception: Type error: cannot unify num with (TFun (TVar 3) (TVar 3))

> (runType '(fun (f) (fun (x) (f (f x)))))
(TFun (TFun (TVar 3) (TVar 3)) (TFun (TVar 3) (TVar 3)))
```

Nota: Los mensajes de error de su implementación deben ser tal y como se muestran en estos ejemplos. Para esto revise la función `error` de racket y utilice la función `prettyfy` provista en base.rkt para obtener la representación en `String` de un `Type` dado.