Tarea 1

Rodrigo Delgado
IC3004 Lenguajes de Programacion

1 Tarea 1

Esta tarea se distribuye con dos ficheros *base.rkt* y *tests.rkt* (disponibles en ucampus). Considere las definiciones del archivo *base.rkt* y escriba sus funciones en el. Escriba sus tests en el archivo *tests.rkt*. Ambos ficheros deben ser entregados vía ucampus. Los tests forman parte de su evaluación!.

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 este bien tipado sin requerir especificaciones de tipos (Si, como python).

En particular, el sistema de tipos pedidos 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 siguiente expresión `(fun (x) (+ x 1))`, el inferenciador lo que hace (muy informalmente) es lo siguiente:

1. Genera una nueva variable de tipo `T1` para denotar el tipo desconocido de x.
2. Genera la *constraint* `T1`=`TNum` para reflejar que esta es la unica forma con que la funcion pueda estar bien tipada.
3. Obtiene que la expresión tiene el tipo `T1` → `TNum` con la *constraint* `T1`=`TNum` generada anteriormente.
4. Aplica la substitución de `T1` por `TNum`.
5. Retorna al usuario que la expresión tiene el tipo `(TNum→ TNum)`.

En las siguientes secciones se muestra paso por paso lo que usted tiene que hacer para poder implementar el inferenciador de tipos

Para esto considere las definiciones adicionales provistas en el archivo *base.rkt*

2 Definiciones

Para representar el tipo de una expresion se usara el siguiente tipo de dato `Type`:

```
(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, una *constraint* `T1=T2` se puede representar simplemente como un par de tipos:

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

3 TypeOf (3.0pt)

Para esta seccion se define el siguiente ambiente para tipos, que asocia un identificador con un tipo (de forma similar a lo visto en clases, en donde se definio un ambiente donde se asocia un identificador con un valor)

```
(deftype TEnv
  (mtTEnv)
  (anTEnv id Type env))
```

Defina la función **lookupT-env :: Sym x TEnv → Type** que dado un identificador y un ambiente de tipos, retorna el tipo asociado al identificador.

Defina la función **typeof :: Expr x TEnv → (Type, List[Constraint])** que dada una expresión `Expr` y un ambiente de tipos `TEnv`, retorna el tipo de la expresión con la lista de *constraints* que debe ser solucionable para que el programa sea valido en tipos. La funcion reporta errores solo en caso de identificadores libres.

Para obtener el tipo de una expresión siga la siguiente tabla:

- 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 funcion consiste en el tipo del argumento (a la izquierda) y en el tipo del cuerpo (a la derecha);

- El tipo de una aplicacion es `(TVar X)` en donde X es un nuevo id
- El tipo del if0 es el tipo de la branch tb

Para obtener la lista de *constraint* siga la siguiente tabla:

```
C(n) = {}
C(x) = {}
C({+ a1 a2}) = C(a1);C(a2);{T1 = TNum};{T2 = Num}  //Donde T1 es el
   tipo de a1 y T2 es el tipo de a2
C({- a1 a2}) = C(a1);C(a2);{T1 = TNum};{T2 = Num}  //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);{T1 = TNum};{T2 = T3}  //Donde
   T1 es el tipo de e, T2 es el tipo de tb y  T3 es el tipo de tf
C({fun param body}) = C(body)
C({app fun arg}) = C(fun-expr);C(arg-expr);(T1 = T2 -> Tx) //Donde
   T1 es el tipo de fun, T2 el tipo de arg, Tx es la variable de
   tipo que representa el cuerpo de la funcion. (Note que el tipo
   de la aplicacion es Tx)
```

3.1 Ejemplos

```
>(typeof  (num 3) (mtTEnv))
(list (TNum))

>(typeof  (add (num 10) (num 3)) (mtTEnv))
(list (TNum) (Cnst (TNum) (TNum)) (Cnst (TNum) (TNum)))

>(typeof  (id 'x) (anTEnv 'x (TNum)  (mtTEnv)))
(list (TNum))

>(typeof  (add (num 10) (id 'x)) (anTEnv 'x (TVar 1)  (mtTEnv)))
(list (TNum) (Cnst (TNum) (TNum)) (Cnst (TVar 1) (TNum)))

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

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

>(typeof  (if0 (num 2) (num 5) (num 3)) (mtTEnv))
(list (TNum) (Cnst (TNum) (TNum)) (Cnst (TNum) (TNum)))
```

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 funcion reset para reiniciar la numeración de *get-id*

4 Unify (2.0pt)

Para saber si existe un *set* de soluciones una vez obtenido la lista de *constraint*, y para obtener el 'mejor' tipo hay que usar el proceso llamado unificacion[1]

Defina primero la funcion **substitute:: TVar x Type x List[Const] → List[Const]** que retorna una lista de *constraints* en donde se reemplazaron todas las ocurrencias de `TVar` en `List[Const]` por `Type`.

Luego defina la funcion **unify :: List[Const] → List[Const]** que dada una lista de *constraints* retorna la lista 'unificada' de *constraints*.

El algoritmo para unificar se describe a continuación[2]:

```
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 y no son variables de tipo (ie. No son TVar)
            unify(C')  //Seguir con el resto de C
        else if T1 es una variables de tipo (y T1 no esta dentro de
           las variables libres 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 esta dentro de
           las variables libres 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} + {T1b = T2b})
        else
            Error
```

5 RunType (1.0pt)

Una vez que se obtiene la lista con las variables de tipos, el *type checker* debe usarla para retornar al usuario un tipo.

Defina la funcion **lookup-list :: List[Constraints] x Tvar → Type** que dada una lista de *constrains* y un tipo `T`, busca en la lista el tipo que esta asociado a la variable de tipo `T`.

[1] https://en.wikipedia.org/wiki/Unification_(computer_science)

[2] Los detalles del algoritmo se encuentra en el libro *Type Systems for Programming Languages*, B. Pierce

Finalmente defina la funcion **runType :: S-Expr → Type** que dada una expresion en syntaxis concreta, retorna su tipo (o arroja un error).

5.1 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 2) (TVar
   2))
```

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 representacion en String de un `Type` dado.