Profesor: Federico Olmedo Ismael Figueroa
Auxiliares: Vicente Gatica Rodrigo Urrea Damian Arquez
Ayudantes: Cristobal Ardiles Rodrigo Iturrieta José Romero
Cristóbal Suazo Cristian Carrión Gaspar Ricci
Lucas Murray Rodrigo Llull

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Tarea 3
Typechecking y CEK machines

Para la resolución de la tarea recuerde que

- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `T3.rkt`) y un conjunto significativo de tests (en `test.rkt`).
- Todo datatype definido por el usuario (via `deftype`) debe estar acompañado de una breve descripción coloquial y de la gramática BNF que lo genera.

Si la función o el datatype no cumple con estas reglas, será ignorado.

El archivo `T3.rkt` contiene una definición inicial del tipo de datos `Expr` con operaciones numéricas y funciones de primera clase. Note que la suma, resta y multiplicación se representan por un único nodo de la sintaxis abstracta: `binop`, que además de contener las sub-expresiones, guarda el operador correspondiente (ver la implementación de `parse`).

Ejercicio 1 **30 Pt**

El *chequeo de tipos estático* es una técnica que permite anticipar ciertas clases de errores, sin necesidad de ejecutar o evaluar un programa. En los intérpretes de las tareas anteriores, cuando un valor no poseía el tipo correcto (por ejemplo, al sumar un booleano con un número), un error se levantaba **durante la evaluación** del programa. En otras palabras, todos nuestros intérpretes, hasta ahora, se han caracterizado por el *chequeo de tipos dinámico*.

Aunque el chequeo dinámico permite, por ejemplo, desarrollar código más rápidamente, hacerlo estáticamente otorga feedback de manera más temprana[1] y permite eliminar chequeos innecesarios durante la ejecución, mejorando la eficiencia.

Un *juicio* de chequeo de tipos $\Gamma \vdash e : T$ se lee "la expresión e tiene tipo T, bajo las premisas contenidas en Γ", donde Γ es un ambiente que relaciona variables con sus tipos. Por ejemplo, el siguiente es un juicio válido de chequeo de tipos:

$$\underbrace{(x : \mathsf{Number})}_{\Gamma} \vdash \underbrace{1 + x}_{e} : \underbrace{\mathsf{Number}}_{T}$$

Dado que x es de tipo Number, es válido decir que la expresión $1 + x$ tiene tipo Number.

(a) [6 Pt] Por ahora solo trabajaremos con dos clases de tipos:
- `Number`: Tipo de los números y expresiones numéricas.

[1] Incluso más si su IDE posee chequeo de tipos en tiempo real.

- `(-> T1 T2)`: Tipo de funciones que reciben un argumento de tipo `T1` y retorna un valor de tipo `T2`.

Defina el tipo de datos `Type` (con su gramática BNF) y la función `parse-type` que recibe una expresión de tipo `s-expr` y retorna un `Type`.

```
>>> (parse-type 'Number)
(numT)
>>> (parse-type '(-> Number Number))
(arrowT (numT) (numT))
>>> (parse-type '(-> (-> Number Number) Number))
(arrowT (arrowT (numT) (numT)) (numT))
```

Una forma de modelar un juicio de chequeo de tipos es usando una función parcial[2], `infer-type`, que recibe como argumentos una expresión e y un ambiente de tipos Γ, y retorna el tipo T de la expresión o levanta un error especificando una violación de tipos.

(b) [14 Pt] Defina la función `infer-type` que opere de la siguiente manera:

- **Literales numéricos**: De cualquier constante numérica se infiere inmediatamente que tiene tipo `Number`.

$$\text{NUM}\ \frac{}{\Gamma \vdash n : \mathsf{Number}}$$

- **Adiciones**: Si ambos operandos tienen tipo `Number`, entonces se infiere que la suma tiene tipo `Number`. Note que:
  - Si alguno de los operandos no tiene el tipo esperado, se debe lanzar un error.
  - Los operandos pueden ser arbitrariamente grandes, es decir, no necesariamente son literales numéricos.

$$\text{SUM}\ \frac{\Gamma \vdash e_1 : \mathsf{Number} \qquad \Gamma \vdash e_2 : \mathsf{Number}}{\Gamma \vdash (+\ e_1\ e_2) : \mathsf{Number}}$$

  Las reglas para la resta y multiplicación son análogas.

- **Identificadores**: Se extrae su tipo desde el ambiente Γ.

$$\text{VAR}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

- **Funciones**: Se debe inferir el tipo del cuerpo bajo un ambiente de tipos extendido con la variable que la función está introduciendo. Luego, el tipo de la función es una flecha con dominio y codominio correspondientes a los

[2] Una función parcial es aquella que puede no retornar un valor en todos los casos porque, por ejemplo, levantó un error.

tipos de su variable y su cuerpo, respectivamente.

$$\text{FN}\ \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\mathsf{fun}\ (x : T_1)\ e) : (\rightarrow\ T_1\ T_2)}$$

- **Aplicaciones**: Se debe obtener el tipo de la expresión en posición de función y verificar que tenga tipo flecha. Luego, se debe obtener el tipo de la expresión en posición de argumento y verificar que su tipo sea igual al esperado por la función:

$$\text{APP}\ \frac{\Gamma \vdash e_1 : (\rightarrow\ T_1\ T_2) \qquad \Gamma \vdash e_2 : T \qquad T = T_1}{\Gamma \vdash (e_1\ e_2) : T_2}$$

```
>>> (infer-type (num 1) empty-tenv)
(numT)

>>> (infer-type (fun 'x (numT) (id 'x)) empty-tenv)
(arrowT (numT) (numT))

>>> (infer-type (fun 'x (arrowT (numT) (numT)) (id 'x)) empty-tenv)
(arrowT (arrowT (numT) (numT)) (arrowT (numT) (numT)))

>>> (infer-type (binop '+ (num 1) (fun 'x (numT) (id 'x))) empty-tenv)
infer-type: invalid operand type for +

>>> (infer-type (app (num 1) (num 2)) empty-tenv)
infer-type: function application to a non-function

>>> (infer-type (app (fun 'x (numT) (id 'x)) (fun 'x (numT) (id 'x)))
    empty-tenv)
infer-type: function argument type mismatch
```

(c) [10 Pt] Extienda el lenguaje (sintaxis concreta, abstracta y las funciones de parsing) con valores booleanos, la operación menor-o-igual `<=`, y la expresión `(if c th el)`, análogos a la Tarea 2. También debe extender la función `infer-type` para soportar estas nuevas expresiones. Las reglas de inferencia de tipos son análogas a las vistas anteriormente. De todas maneras, las reglas para menor-o-igual e if son las siguientes:

$$\text{LEQ}\ \frac{\Gamma \vdash e_1 : \mathsf{Number} \qquad \Gamma \vdash e_2 : \mathsf{Number}}{\Gamma \vdash (<=\ e_1\ e_2) : \mathsf{Boolean}}$$

$$\text{IF}\ \frac{\Gamma \vdash e_1 : \mathsf{Boolean} \qquad \Gamma \vdash e_2 : T_2 \qquad \Gamma \vdash e_3 : T_3 \qquad T_2 = T_3}{\Gamma \vdash (\mathsf{if}\ e_1\ e_2\ e_3) : T_2}$$

```
>>> (parse-type 'Boolean)
(boolT)

>>> (parse '(if (<= 5 6) true false))
(ifc (binop '<= (num 5) (num 6)) (tt) (ff))

>>> (infer-type (ifc (binop '<= (num 5) (num 6)) (num 2) (num 3))
    empty-tenv)
(numT)

>>> (infer-type (ifc (num 5) (num 2) (num 3)) empty-tenv)
infer-type: if condition must be a boolean

>>> (infer-type (ifc (binop '<= (num 5) (num 6)) (num 2) (tt)) empty-tenv)
infer-type: if branches type mismatch
```

IMPORTANTE: En esta pregunta solo debe considerar el lenguaje inicial, no la versión extendida en P1.c. Solo debe trabajar con funciones y operaciones numéricas.

Ejercicio 2 **30 Pt**

Hasta ahora, a lo largo del curso, hemos trabajado exclusivamente con el concepto de *tree-walk interpreters*, es decir, intérpretes que recorren el árbol de una expresión (AST) reduciendo recursivamente sus ramas, hasta llegar a un único nodo con un valor. Por ejemplo, considere la siguiente linea para evaluar una suma:

```
[(add l r) (num+ (eval l env) (eval r env))]
```

Note que las sub-expresiones `l` y `r` son evaluadas recursivamente.

Otra forma de escribir un intérprete es usando maquinas de estado, donde la reducción se define por un solo paso (de un estado a otro) y la evaluación de una expresión depende de hacer avanzar el estado (iterativamente) hasta llegar a un estado final.[3]

En este ejercicio implementaremos un intérprete para nuestro lenguaje utilizando una maquina CEK.

El estado de una maquina CEK consiste en 3 componentes:

- **C**ontrol: la expresión por ser evaluada.
- **E**nvironment: el ambiente de sustitución diferida, que representaremos por el simbolo γ. Note eso sí, que en la maquina CEK, el ambiente no guarda valores, sino que un par *(valor, ambiente)* (clausuras de valores).
- **K**ontinuation: un stack de acciones por realizar, esperando que el término *control* llegue a un valor.

(a) [2 Pt] En las tareas anteriores, definimos el tipo de datos `Val`. En contraste, en esta tarea tomaremos otro enfoque. Defina la función `final?` que dada una expresión (de tipo `Expr`), retorna un booleano indicando si la expresión es un valor o no.

```
>>> (final? (num 1))
#t
>>> (final? (fun 'x (numT) (id 'x)))
#t
>>> (final? (binop '+ (num 1) (num 2)))
#f
```

(b) [6 Pt] Defina el tipo de datos `Kont` con los siguientes constructores:

- `mt-k`: Es la continuación vacía. No contiene información y solo se utiliza al iniciar una evaluación.

- `binop-r-k`: Esta continuación contiene el componente derecho (sin evaluar) de una operación binaria, el ambiente en que se debe evaluar, y una referencia a la continuación anterior. De esta continuación se puede inferir que el componente **C** (control) es el componente izquierdo siendo evaluado.

[3] El estado de una maquina puede estar representado por diversas estructuras, y no se debe confundir con la noción de estado de, por ejemplo, un programa con mutaciones.

- `binop-l-k`: Esta continuación contiene el componente izquierdo **ya evaluado** de una operación binaria, el ambiente en que se evaluó, y una referencia a la continuación anterior. De esta continuación se puede inferir que el componente **C** (control) es el componente derecho siendo evaluado.

- `arg-k`: Esta continuación contiene el argumento (no evaluado) de una función, el ambiente en que éste debe evaluarse, y una referencia a la continuación anterior. De esta continuación se puede inferir que el componente **C** (control) es la función siendo evaluada.

- `fun-k`: Esta continuación contiene una función **ya evaluada**, el ambiente en que fue evaluada, y una referencia a la continuación anterior. De esta continuación se puede inferir que el componente **C** (control) es el argumento siendo evaluado.

(c) [2 Pt] Defina el tipo de datos `State` que tiene un solo constructor `st` que guarda los tres componentes de la máquina CEK: control, environment y kont.

(d) [2 Pt] Defina la función `inject` que recibe una expresión y crea un estado inicial, con un ambiente vacío y la continuación vacía.

(e) [16 Pt] Defina la función `step` que recibe un estado de la maquina CEK, y produce uno nuevo. Esta función **no puede** ser recursiva ni llamar a `eval`: solo debe ejecutar **un paso** de reducción de acuerdo a la siguiente especificación:

$$\text{(RLEFT)} \quad ((\texttt{op}\ e_1\ e_2), \gamma, k) \mapsto (e_1, \gamma, \texttt{binop-r-k}(\texttt{op}, e_2, \gamma, k))$$

$$\text{(RVAR)} \quad (x, \gamma, k) \mapsto (v, \gamma', k) \quad \text{donde } (v, \gamma') = \gamma(x)$$

$$\text{(RFUN)} \quad ((e_1\ e_2), \gamma, k) \mapsto (e_1, \gamma, \texttt{arg-k}(e_2, \gamma, k))$$

$$\text{(RRIGHT)} \quad (v_1, \gamma, \texttt{binop-r-k}(\texttt{op}, e_2, \gamma', k)) \mapsto (e_2, \gamma', \texttt{binop-l-k}(\texttt{op}, v_1, \gamma, k))$$

$$\text{(RBINOP)} \quad (v_2, \gamma, \texttt{binop-l-k}(\texttt{op}, v_1, \gamma', k)) \mapsto (v_1 [\![\texttt{op}]\!] v_2, \gamma, k)$$

$$\text{(RARG)} \quad (v_1, \gamma, \texttt{arg-k}(e_2, \gamma', k)) \mapsto (e_2, \gamma', \texttt{fun-k}(v_1, \gamma, k))$$

$$\text{(RAPP)} \quad (v_2, \gamma, \texttt{fun-k}((\texttt{fun}\ (x : \_)\ e), \gamma', k)) \mapsto (e, \gamma'[x \mapsto (v_2, \gamma)], k)$$

La notación $v_1 [\![\texttt{op}]\!] v_2$ indica la realización de la operación. Por ejemplo si $v_1 = (\texttt{num}\ 1)$, $v_2 = (\texttt{num}\ 2)$ y $\texttt{op} = \texttt{'+}$, entonces $v_1 [\![\texttt{op}]\!] v_2 = (\texttt{num}\ 3)$.

El siguiente ejemplo muestra la consecutiva aplicación de `step` a la expresión `(+ 1 2)`, para reducir finalmente a `3`:

```
>>> (step (st (binop '+ (num 1) (num 2)) (mtEnv) (mt-k)))
(st (num 1) (mtEnv) (binop-r-k '+ (num 2) (mtEnv) (mt-k))) ; (Rleft)

>>> (step (st (num 1) (mtEnv) (binop-r-k '+ (num 2) (mtEnv) (mt-k))))
(st (num 2) (mtEnv) (binop-l-k '+ (num 1) (mtEnv) (mt-k))) ; (Rright)

>>> (step (st (num 2) (mtEnv) (binop-l-k '+ (num 1) (mtEnv) (mt-k))))
(st (num 3) (mtEnv) (mt-k)) ; (Rbinop)
```

El siguiente ejemplo muestra la consecutiva aplicación de `step` a la expresión `((fun (x : Number)x)2)`, para reducir finalmente a `2`:

```
>>> (step (st (app (fun 'x (numT) (id 'x)) (num 2))
               (mtEnv)
               (mt-k)))
(st (fun 'x (numT) (id 'x)) (mtEnv) (arg-k (num 2)
    (mtEnv)
    (mt-k))) ; (Rfun)

>>> (step (st (fun 'x (numT) (id 'x)) (mtEnv) (arg-k (num 2)
              (mtEnv)
              (mt-k))))
(st (num 2)
    (mtEnv)
    (fun-k (fun 'x (numT) (id 'x)) (mtEnv) (mt-k))) ; (Rarg)

>>> (step (st (num 2)
          (mtEnv)
          (fun-k (fun 'x (numT) (id 'x)) (mtEnv) (mt-k))))
(st (id 'x)
    (extend-env 'x (cons (num 2) (mtEnv)) (mtEnv))
    (mt-k)) ; (Rapp)

>>> (step (st (id 'x)
              (extend-env 'x (cons (num 2) (mtEnv)) (mtEnv))
              (mt-k)))
(st (num 2) (mtEnv) (mt-k)) ; (Rvar)
```

(f) [2 Pt] Finalmente, defina la función `run` que recibe una expresión `s-expr`, la parsea, y retorna un par con la expresión evaluada y su tipo.

```
>>> (run '(+ 1 2))
(cons (num 3) (numT))
```