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 2
Parsing e Intérpretes

Para la resolución de la tarea recuerde que

- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `T2.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.

Ejercicio 1 **46 Pt**

El archivo `T2.rkt` contiene una definición inicial del tipo de datos recursivo `Expr` con constantes numéricas, adición, resta y multiplicación.

(a) [4 Pt] Extienda `Expr` con expresiones booleanas, e implemente la función `parse` de tipo `s-expr -> Expr`. En particular, agregue 4 nuevos constructores:

- `(tt)` y `(ff)`: Representan los valores `true` y `false` del lenguaje.

```
>>> (parse true)
(tt)
>>> (parse false)
(ff)
```

- `(leq expr expr)`: Operación menor-o-igual.

```
>>> (parse '(<= 3 5))
(leq (num 3) (num 5))
```

- `(ifc expr expr expr)`: Expresión `if` donde la primera expresión es evalua a un booleano, y la segunda y tercera expresión corresponde a la rama *then* y *else*, respectivamente.

```
>>> (parse '(if (<= 3 5) 2 4))
(ifc (leq (num 3) (num 5)) (num 2) (num 4))
```

(b) [8 Pt] Extienda la gramática de su lenguaje y la función `parse` con funciones de primera clases de múltiples argumentos. Para esto, extienda `Expr` con 3 nuevas expresiones:

- `x`: identificadores o variables.

```
>>> (parse 'x)
(id 'x)
```

- `(fun (id ... ) e)`: funciones con múltiples parámetros.

```
>>> (parse '(fun (x y) (+ x y)))
(fun ( list 'x 'y) (add (id 'x) (id 'y)))
```

- `(f e ... )`: aplicación con múltiples argumentos.

```
>>> (parse '(my-function 2 3 4))
(app (id 'my-function) ( list (num 2) (num 3) (num 4)))
```

Importante. Para hacer pattern matching sobre una lista de cualquier cantidad de elementos puede utilizar la *elipsis* (escrito ...) de la siguiente manera:

```
>>> (match '(2 3 4 5)
       [( list x elems ... ) ( first elems)]) ; x captura el 2
3
>>> (match '(2 (3 4 5))
       [( list x (y rest ... )) (+ x (last rest))]) ; x captura 2, y captura 3
7
```

En el primer ejemplo, la variable `elems` es una lista que contiene todos los elementos excepto el 2. En el segundo ejemplo, la variable `rest` es una lista que contiene los números 4 y 5.

(c) [2 Pt] Defina el tipo de datos recursivo `Val` que capture la noción de valores del lenguaje, y escriba su gramática. Al igual que en clases, deseamos mantener la noción de *scope estático*. Por lo tanto, dado que tenemos funciones de primer orden, es necesario implementar clausuras. Tome en cuenta esto cuando defina los valores del lenguaje.

(d) [6 Pt] En clases se vio la definición de la función `num+`, que realiza la suma subyacente de los valores. Dado que en nuestro lenguaje además tenemos las operaciones `-`, `*` y `<=`, definiremos dos funciones de orden superior:

- `num2num-op` de tipo `(Number Number -> Number)-> (Val Val -> Val)`: Recibe una función que realiza una operación binaria sobre dos números (retornando un número) y retorna una función que recibe dos valores, hace pattern matching sobre los constructores para acceder a los números, y aplica la operación, retornando un nuevo valor. Si los dos valores que recibe no son de carácter numérico, la función debe fallar.

- `num2bool-op` de tipo `(Number Number -> Boolean)-> (Val Val -> Val)`: Análoga a `num2num-op`, pero retorna un valor de carácter booleano.

Luego, defina sus funciones auxiliares de la siguiente manera:

```
(define num+ (num2num-op +))
(define num- (num2num-op -))
(define num* (num2num-op *))
(define num<= (num2bool-op <=))
```

```
>>> (num+ (numV 3) (numV 4))
```

```
(numV 7)
>>> (num+ (numV 4) (boolV #t))
num-op: invalid operands
```

(e) [14 Pt] Implemente la función `eval`, de tipo `Expr Env -> Val`.

(f) [6 Pt] Extienda su gramática y parser con tuplas de múltiples expresiones:

- `(tuple expr ... )`: Crea una tupla con las respectivas expresiones.

```
>>> (parse '(tuple 1 2 3))
(tupl ( list  (num 1) (num 2) (num 3)))
```

- `(proj expr expr)`: Calcula la proyección del componente i-ésimo de la tupla.

```
>>> (parse '(proj (tuple 10 20 30) 1))
(proj (tupl ( list  (num 10) (num 20) (num 30))) (num 1))
```

(g) [6 Pt] Un tupla se considera un valor cuando todos sus componentes internos son valores. Actualice su definición de `Val` y extienda la función `eval` para soportar tuplas y sus proyecciones.

Ejercicio 2 **14 Pt**

Ahora implementaremos funciones globales para que podamos usar dentro de nuestro lenguaje. Dado que nuestro intérprete ya recibe un ambiente de valores, es directo definir clausuras que podemos inyectar al inicio de la interpretación. Por ejemplo, el siguiente código utiliza una función global `id` (una función que simplemente retorna su argumento):

```
(define id*
    (closureV '(x) (parse 'x) empty-env)) ;; <- Definimos la funcion global

(define my-program (parse '{+ 1 (id 2)}))

(eval my-program
    (extend-env 'id id* empty-env)) ;; <- Inyectamos 'id' en el ambiente inicial
```

(a) [8 Pt] **Utilizando su lenguaje**, al igual que en el ejemplo anterior, implemente las siguientes funciones:

- `swap*`: Recibe una función de dos argumentos, de tipo `a b -> c`, y crea una nueva función que recibe los argumentos en el orden contrario, es decir, de tipo `b a -> c`.

- `curry*`: Recibe una función de tipo `a b -> c` y retorna su versión currificada, de tipo `a -> b -> c`

- `uncurry*`: Recibe una función de tipo `a -> b -> c` y retorna su versión no currificada, de tipo `a b -> c`.

- `partial*`: Recibe una función de tipo `a b -> c` y un argumento de tipo `a`, y retorna una función de tipo `b -> c`.

(b) [6 Pt] Defina la función `run` de tipo `s-expr (Listof (Pair Symbol Val))-> Val` que recibe una expresión fuente y una lista de pares, donde cada par contiene el nombre de la variable global y su valor asociado. La función debe evaluar el programa inyectando los valores globales.

```
(define globals (list
                 (cons 'swap swap*)
                 (cons 'curry curry*)
                 (cons 'uncurry uncurry*)
                 (cons 'partial  partial*))

>>> (run
       '((swap (fun (x y) (<= x y))) 1 2)
       globals)
(boolV #f)
>>> (run
       '((partial (fun (x y) (<= x y)) 1) 2)
       globals)
(boolV #t)
```