Documentación del lenguaje de programación Fennel

Integrantes del equipo:

1. Castro Meza, Carlos Alberto
2. Fernandez Gutierrez, Bruno Raúl
3. Pino Zavala, Joaquín Felipe
4. Quispe Neira, Fredy Goolberth

Índice:

1. Locales y variables

Fennel es un lenguaje de programación dinámicamente tipado, lo cual hace que el tipo de las variables definidas se establezca al momento de ejecución del programa.

Las locales se definen con la palabra reservada “let” teniendo el nombre de la variable y los valores dentro de corchetes. Se tiene la posibilidad también, de declarar varios locales en un mismo corchete.

Ejemplo:

En este caso la local “x” está tomando el valor de la operación (89 + 5.2), mientras que la local “f” está siendo definida como una función que imprime el doble del argumento pasado a la función. Estas locales solo son válidas dentro del bloque “let”.

```
(let [x (+ 89 5.2)
      f (fn [abc] (print (* 2 abc)))]
  (f x))
```

Las locales también pueden ser definidas mediante la palabra reservada “local”. Esto se usa cuando se requiere usar la local en todo el programa.

Ejemplo:

```
(local tau-approx 6.28318)
```

Las locales no pueden tomar nuevos valores. Para este propósito se tiene 2 opciones: Se pueden declarar nuevas locales internas en bloques “let” internos para definir una nueva local con el mismo nombre.

Ejemplo:

Se puede ver en el siguiente ejemplo cómo se redefine la local“x” en otro bloque interno “let” dentro del primer bloque. Es así cómo podemos “cambiar” el valor de una local sobreescribiendo (shadowing) otra local.

```
(let [x 19]
  ;; (set x 88) <- not allowed!
  (let [x 88]
    (print (+ x 2))) ; -> 90
  (print x)) ; -> 19
```

La segunda opción para poder dar a una local otro valor después de su definición es utilizando la palabra reservada “var”. Está palabra permite asignar un nuevo valor a la variable con la función “set”. En este caso la funcionalidad de “var” no cuenta con la capacidad de bloques internos como “let”.

Ejemplo:

En este ejemplo se permite modificar el valor de la variable “x” mediante la palabra clave “set”.

```
(var x 19)
(set x (+ x 8))
(print x) ; -> 27
```

2. Números y strings

Todos los operadores aritméticos estándar como +,-,* y / trabajan en forma prefija. En la versión de Lua utilizada, 5.1, los números son floats de doble precisión. Para valores grandes se puede utilizar el guión bajo, el cual no tendrá ningún efecto adverso en la salida.

```
(let [x (+ 1 99)
      y (- x 12)
      z 100_000]
  (+ z (/ y 10)))
```

Los strings son esencialmente arrays de bytes inmutables. El soporte de UTF-8 es proporcionado por una librería externa. Los strings se concatenan con .. de la siguiente manera:

```
(.. "hello" " world")
```

3. Estructuras de datos

3.1 Tablas

Como tal, la sección de estructuras de datos solo está comprendida por las tablas, ya que al usar Lua, Fennel también cuenta con esta característica. La sintaxis principal de las tablas utiliza llaves con pares clave/valor:

```
{"key" value
 "number" 531
 "f" (fn [x] (+ x 2))}
```

También es posible usar el "." para obtener valores de la tabla:

```
(let [tbl (function-which-returns-a-table)
      key "a certain key"]
  (. tbl key))
```

A su vez también podemos usar "tset" para insertarlos:

```
(let [tbl {}
      key1 "a long string"
      key2 12]
  (tset tbl key1 "the first value")
  (tset tbl key2 "the second one")
  tbl) ; -> {"a long string" "the first value" 12 "the second one"}
```

3.2 Tablas secuenciales

También existen tablas que almacenan datos de forma secuencial. Aquí, las llaves empiezan en 1 y van aumentando. La sintaxis de Fennel para estas estructuras usa corchetes.

```
["abc" "def" "xyz"] ; equivalent to {1 "abc" 2 "def" 3 "xyz"}
```

La función table.insert de Lua se usa para estas tablas secuenciales. Todos los valores después del valor insertado se desplazan un índice hacia arriba. Si no se proporciona un índice, se agregará al final.

La función table.remove funciona de manera similar; toma una tabla y un índice (que por defecto es el final de la tabla) y elimina el valor en ese índice, devolviéndolo.

```
(local ltrs ["a" "b" "c" "d"])

(table.remove ltrs)       ; Removes "d"
(table.remove ltrs 1)     ; Removes "a"
(table.insert ltrs "d")   ; Appends "d"
(table.insert ltrs 1 "a") ; Prepends "a"

(. ltrs 2)                ; -> "b"
;; ltrs is back to its original value ["a" "b" "c" "d"]
```

Length retorna la longitud de tablas secuenciales y strings.

```
(let [tbl ["abc" "def" "xyz"]]
  (+ (length tbl)
     (length (. tbl 1)))) ; -> 6
```

4. Bucles

4.1 Bucle Each

El bucle sobre los elementos de la tabla se realiza con “each” y un iterador como “pairs” (usados para tablas generales) o “ipairs” (para tablas secuenciales):

```
(each [key value (pairs {"key1" 52 "key2" 99})]
  (print key value))

(each [index value (ipairs ["abc" "def" "xyz"])]
  (print index value))
```

Se puede llamar a “ipairs” en cualquier tabla, y sólo iterará sobre claves numéricas comenzando con 1 hasta llegar a cero.

4.2 Bucles For y While

El bucle “for” itera numéricamente desde el valor inicial proporcionado hasta el valor final, incluyéndolo.

```
(for [i 1 10]
  (print i))
```

Se puede especificar un valor de paso opcional; este bucle sólo imprimirá números impares menores de diez:

```
(for [i 1 10 2]
  (print i))
```

Si necesitamos hacer un bucle sin saber cuántas veces iterar, usamos “while”:

```
(while (keep-looping?)
  (do-something))
```

5. Condicionales

En el caso de este lenguaje se tienen 2 tipos de condicionales: “if” y “when”.

5.1 Condicional If

La palabra reservada “if” tiene una funcionalidad similar a la implementación común en la mayoría de los lenguajes de programación.
Este tipo de bloque también puede ser usado para expandir diferentes casos de expresiones lógicas. En caso de tener un número impar de condiciones en el bloque, la última condición se considera como el caso “else”.

```
(let [x (math.random 64)]
  (if (= 0 (% x 2))
      "even"
      (= 0 (% x 9))
      "multiple of nine"
      "I dunno, something else"))
```

En el caso del lenguaje de Fennel, el bloque “if” retorna una expresión, ya que Fennel tiene base en lisp que no considera sentencias en sí.

5.2 Condicional When

El segundo tipo de condicional de Fennel es “when”, el cual permite realizar múltiples operaciones cuando una condición se evalúa como verdadera.
Tener en cuenta que a diferencia de “if”, este tipo de condicional no permite múltiples condicionales o caso “else”.

```
(when (currently-raining?)
  (wear "boots")
  (deploy-umbrella))
```

6. Funciones

6.1 Funciones comunes

Para definir una función en Fennel se usa la palabra reservada “fn”, al momento de definir una función se puede colocar un nombre para está de manera opcional, el cual estará enlazado a la función en un ámbito local. En caso que no se especifique el nombre, simplemente tendrá un valor anónimo.

En cuanto a la sintaxis general para la definición de una función en Fennel se deben colocar los argumentos destinados a esta entre corchetes “[]”.

Ejemplo:

En este caso, se crea una función llamada “print-and-add” que recibe como parámetros las variables [a b c]. Como su nombre indica, la función imprime el valor de la variable “a” y además suma las variables “b” y “c”.

```
(fn print-and-add [a b c]
  (print a)
  (+ b c))
```

6.2 Funciones lambda

Fennel provee la opción de usar funciones “lambda”. A pesar de que las funciones comunes pueden tener un rendimiento mayor, estas no cuentan con verificación de número de argumentos. Para tener un código más seguro, se puede usar la palabra reservada “lambda” para asegurarse que el número de argumentos es al menos el número que se definió en la función. Se tiene también la posibilidad de tener un argumento opcional, colocando “?” a la izquierda de un argumento.

Ejemplo:

Se puede ver que el argumento “y” es ignorado y puede ser nulo, sin embargo, z daría un error, ya que se está usando una función lambda.

```
(lambda print-calculation [x ?y z]
  (print (- x (* (or ?y 1) z))))

(print-calculation 5) ; -> error: Missing argument z
```

6.3 Funciones variádicas

Este tipo de funciones permiten definir múltiples argumentos con el símbolo "...". Este tipo de argumentos no se debe tratar como una lista, aunque se podría acceder cada uno de ellos similarmente como una lista de argumentos usando () para encapsular estos.

Ejemplo:

En este ejemplo se tiene una función llamada print-each que recibe un número variable de argumentos. Está función imprime cada argumento utilizando la función "ipairs" para poder acceder a la lista de argumentos como una lista. Posteriormente se imprime el índice y argumento recibido en la función. El Símbolo de ".." se utiliza para la concatenación de strings.

```
(fn print-each [...]
  (each [i v (ipairs [...])]
    (print (.. "Argument " i " is " v))))

(print-each :a :b :c)
```

Se debe tener en cuenta que el ámbito de estas variables es diferente a las variables definidas en las funciones comunes. Estas variables solo son accesibles en la función en la que son creadas.

7. Manejo de errores

Los errores en Lua pueden adoptar dos formas. Las funciones en Lua pueden devolver cualquier cantidad de valores, y la mayoría de las funciones que pueden fallar indicarán el error mediante el uso de dos valores de retorno: “nil” seguido de una cadena de mensaje de error. Se puede interactuar con este estilo de función en Fennel desestructurando con paréntesis en lugar de corchetes:

```
(match (io.open "file")
  ;; when io.open succeeds, it will return a file, but if it fails it will
  ;; return nil and an err-msg string describing why
  f (do (use-file-contents (f:read :*all))
        (f:close))
  (nil err-msg) (print "Could not open file:" err-msg))
```

También se puede escribir una función que devuelva múltiples valores con valores.

```
(fn use-file [filename]
  (if (valid-file-name? filename)
      (open-file filename)
      (values nil (.. "Invalid filename: " filename))))
```

El problema de este tipo de error es que no llega a componerse bien; el estado de error debe propagarse a lo largo de la cadena de llamadas desde el interior al exterior. Para solucionar este problema, se utiliza “error”. Esto finalizará todo el proceso a menos que esté dentro de una llamada protegida, similar a la forma en otros lenguajes donde lanzar una excepción detendrá el programa a menos que esté dentro de un try/catch. Puedes realizar una llamada protegida con “pcall”:

```
(let [(ok? val-or-msg) (pcall potentially-disastrous-call filename)]
  (if ok?
      (print "Got value" val-or-msg)
      (print "Could not get value:" val-or-msg)))
```

La invocación de “pcall” allí significa que se está ejecutando (`potentially-disastrous-call filename`) en modo protegido. “pcall” toma un número arbitrario de argumentos que se pasan a la función. Se puede ver que “pcall” devuelve un valor booleano (`ok?`) para informarle si la llamada tuvo éxito o no, y un segundo valor (`val-or-msg`) que es el valor real si tuvo éxito o un mensaje de error si no.

La función “assert” toma un valor y un mensaje de error; llama a error si el valor es nulo y lo devuelve en caso contrario. Esto se puede usar para convertir fallas de múltiples valores en errores.

```
(let [f (assert (io.open filename))
      contents (f.read f "*all")]
  (f.close f)
  contents)
```

8. Módulos y manejo de archivos

Se puede usar la función “require” para cargar código proveniente de otros archivos.

```
(let [lume (require :lume)
      tbl [52 99 412 654]
      plus (fn [x y] (+ x y))]
  (lume.map tbl (partial plus 2))) ; -> [54 101 414 656]
```

Los módulos en Fennel y Lua son simplemente tablas que contienen funciones y otros valores. El último valor de un archivo Fennel se utilizará como valor de todo el módulo. Técnicamente, puede ser cualquier valor, no solo una tabla, pero usar una tabla es lo más común.

Para requerir un módulo que está en un subdirectorio, se debe tomar el nombre del archivo, reemplazar las barras con puntos, eliminar la extensión y luego pasarlo por “require”. Por ejemplo, un archivo llamado lib/ui/menu.lua se leerá al cargar el módulo lib.ui.menu.
.
Cuando se ejecuta un programa con el comando fennel, se puede llamar a “require” para cargar los módulos Fennel o Lua. Pero en otros contextos (como compilar en Lua y luego usar el comando lua, o en programas que incorporan Lua) no sabrá acerca de los módulos de Fennel. Se necesita instalar el buscador que sabe cómo encontrar archivos .fnl:

```
require("fennel").install()
local mylib = require("mylib") -- will compile and load code in mylib.fnl
```

Una vez que agregue esto, “require” funcionará en archivos Fennel tal como lo hace con Lua.