**Ejercicio 1.1:**

local L L2 Length in

fun {Length Xs}

case Xs of nil then

0

[] H|T then

1 + {Length T}

end

end

L = 1|2|3|4|5|6|nil

L2 = nil

{Browse {Length L}}

{Browse {Length L2}}

end

**Ejercicio 1.2:**

local L Take in

fun {Take Xs N}

if N > 0 then

case Xs of nil then

nil

[] H|T then

H|{Take T N-1}

end

else

nil

end

end

L = [1 2 3 4 5 6]

{Browse {Take L 4}}

end

**Ejercicio 1.3:**

local L Drop in

fun {Drop Xs N}

case Xs of nil then

nil

[] H|T then

if N > 0 then

{Drop T N-1}

else

Xs

end

end

end

L = [1 2 3 4 5 6]

{Browse {Drop L 0}}

end

**Ejercicio 1.4:**

local L L2 Append in

fun {Append Xs Ys}

case Xs of nil then

Ys

[] H|nil then

H|Ys

[] H|T then

H|{Append T Ys}

end

end

L = [1 2 3 4 5 6]

L2 = [7 8]

{Browse {Append L L2}}

end

**Ejercicio 1.5:**

local L X Member in

fun {Member Xs Y}

case Xs of nil then

false

[] H|T then

if (H == X) then

true

else

{Member T X}

end

end

end

L = [1 2 3 4 5 6]

X = 6

{Browse {Member L X}}

end

**Ejercicio 1.6:**

local L X Position in

fun {Position Xs X}

local I in

Pos = 1

{PositionAux Xs X Pos}

end

end

fun {PositionInt Xs X Pos}

case Xs of H|T then

if (H == X) then

Pos

else

{PositionInt T X Pos + 1}

end

end

end

L = [1 2 3 9 5 6]

X = 9

{Browse {Position L X}}

end

**Ejercicio 2**

1. proc {P X Y} local Z in {Q Z U} end end

2. proc {P X Y} local Z in {Q Z Y} end end

3. proc {P X Y} local Z in {P Z Y} end end

1. Referencias externas: Q y U

2. Referencia externa: Q

3. No hay referencias externas

**Ejercicio 3:**

Realizado en Excel.

**Ejercicio 4:**

Traducimos a lenguaje kernel para analizar:

local Test A B C D E F G H I J K L M in

proc {Test X}

case X

of a|Z then {Browse 'case'(1)}

[] f(a) then {Browse 'case'(2)}

[] Y|Z andthen Y==Z then {Browse 'case'(3)}

[] Y|Z then {Browse 'case'(4)}

[] f(Y) then {Browse 'case'(5)}

else {Browse 'case'(6)} end

end

A = [b c a]

B = f(b(3)) #f(Y), Y = b(3)

C = f(a)

D = f(a(3)) #f(Y), Y = a(3)

E = f(d)

F = [a b c]

G = [c a b]

H = a|a

I = '|'(v b)

J = '|'(a a)

K = '|'(b b)

L = '|'(a b c)

M = '|'(a [b c])

{Test A}

{Test B}

{Test C}

{Test D}

{Test E}

{Test F}

{Test G}

{Test H}

{Test I}

{Test J}

{Test K}

{Test L}

{Test M}

end

La Ejecución:

'case'(4)

'case'(5)

'case'(2)

'case'(5)

'case'(5)

'case'(1)

'case'(4)

'case'(1)

'case'(4)

'case'(1)

'case'(3)

'case'(6)

'case'(1)

**Ejercicio 5.1:**

proc {Length Xs N}

case Xs of nil then

N = 0

[] X|Xr then

local N int in

{Length Xr Naux}

N = Naux + 1

end

end

end

**Ejercicio 5.2:**

proc {LengthTail Xs N NTail}

case Xs of nil then

N = NTail

[] X|Xr then

{LengthTail Xs N NTail+1}

end

end

proc {Length Xs N}

{LengthTail Xs N 0}

end

La ventaja de usar "tail recursive" es que no se apila un nuevo frame en el stack al efectuar la llamada, sino que se utiliza el frame actual. De ésta manera el procesador puede efectuar el cálculo de manera más fácil y rápida.

**Ejercicio 6.1:**

*Fold*

local FoldL L Suma in

fun {FoldL L F U}

{FoldLInt L F U U}

end

fun {FoldLInt L F U Ini}

case L

of nil then

{F U Ini}

[] H|T then

{FoldLInt T F {F U H} Ini}

end

end

fun {Suma X Y}

X + Y

end

L = [1 2 3 4 5 6]

{Browse {FoldL L Suma 0}}

end

local FoldR L Suma in

fun {FoldR L F U}

{FoldRInt L F U U}

end

fun {FoldRInt L F U Ini}

case L

of nil then

{F U Ini}

[] H|T then

{F U {FoldRInt T F H Ini}}

end

end

fun {Suma X Y}

X + Y

end

L = [1 2 3 4 5 6]

{Browse {FoldL L Suma 0}}

end

**Ejercicio 6.2:**

local Map L Cuadrado in

fun {Map L F}

case L

of nil then

L

[] H|T then

{F H} | {Map T F}

end

end

fun {Cuadrado X}

X \* X

end

L = [1 2 3 4 5 6]

{Browse {Map L Cuadrado}}

end

**Ejercicio 6.3:**

local Filter L Par in

fun {Filter L F}

case L

of nil then

L

[] H|T then

if {F H} then

H | {Filter T F}

else

{Filter T F}

end

end

end

fun {Par X}

X mod 2 == 0

end

L = [1 2 3 4 5 6]

{Browse {Filter L Par}}

End

**Ejercicio 7:**

Currying es la técnica de traducir una evaluación de argumentos en una evaluación de funciones, es decir, traduce una función que tiene múltiples parámetros a una secuencia de funciones con un único parámetro.

Local Max X Y in

fun {Max X}

fun {$ Y}

If X>Y then X else Y end

end

end

X = 16

Y = 8

{Browse {{Max X} Y}}

end

**Ejercicio 8.1:**

local AsignarValor WaitSome S A B C D E S2 F in

proc {WaitSome Xs}

case Xs of H|T then

{WaitSome T}

{WaitOr H T}

end

end

proc {AsignarValor Xs}

Xs.2.2.2.2.2.1 = 3

end

S = [ A B C D E F]

thread

{Browse 'Esperando'}

{WaitSome S}

{Browse S}

end

thread

{AsignarValor S}

{Browse 'Valor asignado'}

end

end

**Ejercicio 9.1:**

local MyMap SquareLazy L Lout A B C in

MyMap = proc {$ Xs F Res}

case Xs of nil then

Res=nil

[] X|Xr then

local H T in

H = {F X}

T = {MyMap Xr F}

Res = H|T

end

end

end

SquareLazy = proc {$ X Res}

local Trigger in

Trigger = proc {$ Res}

Res = X\*X

end

{ByNeed Trigger Res}

end

end

L = [1 2 3 4]

Lout = {MyMap L SquareLazy}

A = Lout.1

B = Lout.2.1

C = A + B

{Browse C}

end

**Ejercicio 9.2:**

* Reverse1: La función es lazy. Una vez que se necesita el valor, se hace la

reversión de toda la lista.

* Reverse2: La función es lazy y la función interna de transformación

también lo es; así, sólo cuando se requiere el elemento se calcula la reversión

hasta tal punto.

**Ejercicio 10:**

local Canal Puerto Mostrar in

proc {Mostrar Canal}

case Canal of H|T then

{Browse H} {Mostrar T}

end

end

{NewPort Canal Puerto}

thread

{Mostrar Canal}

end

{Send Puerto "Mensaje 1"}

{Send Puerto "Mensaje 2"}

{Send Puerto "Mensaje 3"}

end

**Ejercicio 11:**

local Agente Puerto Canal PuertoValidado CanalValidado EnviarAPValidado in

proc {Agente S}

case S of H|T then

{Browse H}

{Agente T}

end

end

proc {EnviarAPValidado S Validar}

case S of H|T then

if {Validar H} then

{Send PuertoValidado H}

end

{EnviarAPValidado T Validar}

end

end

{NewPort CanalValidado PuertoValidado}

{NewPort Canal Puerto}

thread

{Agente CanalValidado}

end

thread

{EnviarAPValidado Canal Char.isLower}

end

{Send PuertoEntrada "a"}

{Send PuertoEntrada "b"}

{Send PuertoEntrada "c"}

{Send PuertoEntrada "A"}

end

**Ejercicio 12:**

declare X Y Z in

X={NewCell 0}

{Assign X 5}

Y=X # Y refiere a la misma celda que X

{Assign Y 10} #Asignamos 10 a la celda de Y y también a la celda de X

{Browse {Access X} == 10} # True

{Browse X==Y} #True, refieren a la misma celda

Z={NewCell 10} #Ahora Z refiere a una \*nueva\* celda que tiene 10

{Browse Z==Y} #Falso, pues las celdas a las que refieren son distintas

{Browse @X==@Z} #Verdadero, las celdas almacenan el mismo valor

**Ejercicio 13:**

local NewDicc Put Get ContarLetras ContarLetras Split MergeSort FlattenTree TuplaMenor Tuplas in

fun {NewDicc}

nil

end

fun {Put Dicc K V}

case Dicc

of tree(K1 V1 L1 R1) andthen K1==K then

tree(K1 V L1 R1)

[] tree(K1 V1 L1 R1) andthen K<K1 then

tree(K1 V1 {Put L1 K V} R1)

[] tree(K1 V1 L1 R1) andthen K1<K then

tree(K1 V1 L1 {Put R1 K V})

else

tree(K V nil nil)

end

end

fun {Get Dicc K}

case Dicc

of tree(K1 V1 L1 R1) andthen K1==K then

V1

[] tree(K1 V1 L1 R1) andthen K<K1 then

{Get L1 K}

[] tree(K1 V1 L1 R1) andthen K1<K then

{Get R1 K}

else

nil

end

end

fun {ContarLetras S Dicc}

case S of H|T then

local Prev Dicc2 in

Prev = {Get Dicc H}

if Prev == nil then

Dicc2 = {Put Dicc H 1}

else

Dicc2 = {Put Dicc H Prev+1}

end

{ContarLetras T Dicc2}

end

[] H|nil then

local Prev in

Prev = {Get Dicc H}

if Prev == nil then

{Put Dicc H 1}

else

{Put Dicc H Prev+1}

end

end

else

Dicc

end

end

fun {Merge A B LessThan}

if A \= nil andthen B \= nil then

if {LessThan A.1 B.1} then

A.1|{Merge A.2 B LessThan}

else

B.1|{Merge A B.2 LessThan}

end

elseif A\=nil then /\*B esta vacia\*/

A.1|{Merge A.2 B LessThan}

elseif B\=nil then /\*A esta vacia\*/

B.1|{Merge A B.2 LessThan}

else /\*Caso base\*/

nil

end

end

proc {Split L R1 R2}

case L of

H1|H2|T then

local X Y in

R1 = H1|X

R2 = H2|Y

{Split T X Y}

end

else

R1 = L|nil

R2 = nil

end

end

fun {MergeSort List LessThan}

case List of

nil then List

[] [H] then List

else

local R1 R2 in

{Split List R1 R2}

{Merge {MergeSort R1 LessThan} {MergeSort R2 LessThan} LessThan}

end

end

end

fun {FlattenTree Tree}

case Tree

of tree(K1 V1 nil nil) then

pair(K1 V1)

[] tree(K1 V1 L1 nil) then

{FlattenTree L1}|pair(K1 V1)

[] tree(K1 V1 nil R1) then

pair(K1 V1)|{FlattenTree R1}

[] tree(K1 V1 L1 R1) then

{FlattenTree L1}|pair(K1 V1)|{FlattenTree R1}

else nil

end

end

fun {TuplaMenor A B}

if A.2 == B.2 then

A.1 < B.1

else

A.2 < B.2

end

end

Tuplas = {FlattenTree {ContarLetras ['B' 'B' 'A' 'C' 'D' 'A' 'K'] {NewDicc}}}

{Browse Tuplas}

{Browse {MergeSort Tuplas TuplaMenor}}

end