Part 1: Syntactic Sugar

----------Kernel.txt----------

[local ["A" "B"] [A = false() local ["EXU1"] [EXU1 = true() if EXU1 then [skip/BA] else [local ["EXU2"] [EXU2 = B if EXU2 then [skip] else [skip]]]], case A of tree() then [skip] else [case A of false() then [skip] else [case A of true() then [skip] else [skip]]]], local ["A"] [A = 2 local ["EXU1"] [local ["EXU2" "EXU3"] [EXU2 = A, EXU3 = 1, "Eq" "EXU2" "EXU3" "EXU1"], if EXU1 then [skip] else [skip]], local ["EXU1"] [local ["EXU2" "EXU3"] [EXU2 = A local ["EXU5

(// I was not able to finish the kernel because I didn't remember the syntax but later got it when I run sugar.txt file.)

----------Sugar.txt----------

```
// 1) nested if, nested case
local A B in
   A = false
   if true then                  // expression in if-condition
      skip Browse A
   elseif B then                  // elsif can be repeated 0 or more times
      skip Basic
   else                          // else is optional
      skip Basic
   end
   case A
      of tree() then skip Basic
      [] false then skip Basic     // nesting symbol is [] followed by record
      [] true then skip Basic
   else                          // else is optional
      skip Basic
   end
end
```

[Explanation:

Two local variables, A and B, are initialized by the code with the value false. Following that, it checks criteria using nested if statements before executing skip Basic based on the values of A and B. Similar to this, nested case statements execute skip Basic for particular patterns while also checking the value of A. In all scenarios, skip Basic is used because A is initially false and B is not explicitly set.

]

// 2) more expressions; note that applications of primitive binary operators

// ==, <, >, +, -, *, mod must be enclosed in parentheses for hoz

```
local A in
   A = 2
   if (A == 1) then              // expression in condition
      skip Basic
   end
   if (A == (3-1)) then          // nested expression
      skip Browse A
   end
end
```

[Explanation:

A local variable is created and its value is set to 2.

In the first if statement, it determines if A is equal to 1. This condition is false because A is 2, hence the if block's code is not carried out.

In the second if statement, it is determined whether A equals the value of the output of the expression (3-1), which is 2. Given that A is likewise 2, this condition is satisfied, and the code inside the if block is carried out, skipping to the section of the program titled "Browse A."

]

// 3) "in" declaration

```
local T = tree(1:3 2:T) X Y in          // Variable = value, variables
   local tree(1:A 2:B) = T in
      if (1==1) then B = (5-2) Z in       // "local" not necessary
```

```
      skip Browse B
    end
  end
end
```

[Explanation:

It declares a local variable T and gives it a value that consists of a reference to T as well as a tree structure with numbers.

It also declares the two additional local variables X and Y, but does not yet give them any values.

Another local declaration, local tree(1:A 2:B) = T, is contained within this block and looks to split the T tree into two variables, A and B.

Following that, an if statement verifies whether 1 equals 1 (which is always true). If so, it defines a new local variable Z and gives B the value (5-2).

Finally, the program goes to a section called "Browse B."

]

```
// 4) expressions in place of statements
local Fun R in
  Fun = fun {$ X}     // function returns a value (last item of function)
          X         // returned value
        end
  R = {Fun 4}         // Var = Expression
  skip Browse R
end
```

[Explanation:

It declares Fun and R as two local variables.

It specifies a Fun function, which accepts a single argument X and returns the same value X.

It sets the variable R to the outcome of running the Fun function with argument 4. R therefore becomes 4.

The program then moves to a section titled "Browse R," which indicates that it is moving to a different section of the program, possibly to display or use the value of R.

]

// 5) Bind fun

local A B in

  skip Basic

  A = rdc(1:4 2:B 3:(B#B)) // Bind with pattern

  B = (5 + (3 - 4)) // Bind with expression

  skip Browse A

  skip Browse B

  skip Store

end

[Explanation:

A and B, two local variables, are declared.

Using a pattern, it gives A a value.

It assigns a value to B by carrying out some mathematical operations: 5 + (3 - 4) simplifies to 5 - 1, which makes B become 4.

The program skips to a section labeled "Browse A," which indicates that it is going to a different section where it might interact with the value of A.

Additionally, it skips to a section of the program named "Browse B," potentially to perform an action on the value of B.

Finally, it jumps to a section of the program labeled "Store," indicating that it is proceeding to a different area of the program where it may carry out data storage or other operations.

]

[OUTPUT]

ghci> runFull "declarative" "sugar.txt" "sugar2kern.txt"

A : false()

A : 2

B : 3

R : 4

A : rdc(1:40 2:41 3:42)

B : 4

Store : ((47), 3),
((48), 4),
((45), 5),
((46), -1),
((44, 43, 41, 39), 4),
((40), 4),
((42), '#'(1:43 2:44)),
((38), rdc(1:40 2:41 3:42)),
((36, 37), 4),
((35), proc(["X","EXU1"],[EXU1 = X],[])),
((33), 5),
((34), 2),
((32), 3),
((31), Unbound),
((29), 1),
((30), 1),
((28), true()),
((26, 27, 24, 22), tree(1:25 2:26)),
((25, 23), 3),
((20), Unbound),
((21), Unbound),
((18), 3),

((19), 1),

((16, 13, 11), 2),

((17), 2),

((15), true()),

((14), 1),

((12), false()),

((10), true()),

((8), false()),

((9), Unbound),

((1), Primitive Operation),

((2), Primitive Operation),

((3), Primitive Operation),

((4), Primitive Operation),

((5), Primitive Operation),

((6), Primitive Operation),

((7), Primitive Operation)

Part 2: Lists and Difference Lists

A. ----------Append.txt----------

```
local Append L1 L2 Out Reverse Out1 in

  // Append function on p 133 (modified for hoz)
  fun {Append Ls Ms}
    case Ls
    of nil then Ms
    [] '|'(1:X 2:Lr) then (X|{Append Lr Ms})
    end
  end

  L1 = (1|(2|(3|nil)))
  L2 = (4|(5|(6|nil)))

  Out = {Append L1 L2}
  skip Browse Out
  skip Full

  // O(n^2) Reverse function on p 135 (modified for hoz):
  fun {Reverse Xs}
    case Xs
    of nil then nil
    [] '|'(1:X 2:Xr) then
      {Append {Reverse Xr} [X]}
    end
  end
```

[OUTPUT]

ghci> runFull "declarative" "append.txt" "append2kern.txt"

Out : [1 2 3 4 5 6]

Store : ((37, 39, 35, 31, 27, 10), '|'(1:20 2:21)),

((38, 19), nil()),

((36, 18), 3),

((34, 17), '|'(1:18 2:19)),

((32, 16), 2),

((33), '|'(1:36 2:37)),

((30, 15), '|'(1:16 2:17)),

((28, 14), 1),

((29), '|'(1:32 2:33)),

((26, 9), '|'(1:14 2:15)),

((24), 6),

((25), nil()),

((22), 5),

((23), '|'(1:24 2:25)),

((20), 4),

((21), '|'(1:22 2:23)),

((8), proc(["Ls","Ms","EXU1"],[case Ls of nil() then [EXU1 = Ms] else [case Ls of '|'(1:X 2:Lr) then [local ["EXU2","EXU3"] [EXU2 = X,local ["EXU4","EXU5"] [EXU4 = Lr,EXU5 = Ms,"Append" "EXU4" "EXU5" "EXU3"],EXU1 = '|'(1:EXU2 2:EXU3)]] else [skip]]],[("Append",8)])),

((11), '|'(1:28 2:29)),

((12), Unbound),

((13), Unbound),

((1), Primitive Operation),

((2), Primitive Operation),

((3), Primitive Operation),

((4), Primitive Operation),

((5), Primitive Operation),

((6), Primitive Operation),

((7), Primitive Operation)

Mutable Store: Empty

Current Environment : ("Append" -> 8, "L1" -> 9, "L2" -> 10, "Out" -> 11, "Reverse" -> 12, "Out1" -> 13, "IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4, "LT" -> 5, "Mod" -> 6, "IntMultiply" -> 7)

Stack : "Reverse = proc {$ Xs EXU1} [case Xs of nil() then [EXU1 = nil()] else [case Xs of '|'(1:X 2:Xr) then [local [\"EXU2\",\"EXU3\"] [local [\"EXU4\"] [EXU4 = Xr,\"Reverse\" \"EXU4\" \"EXU2\"],local [\"EXU4\"] [EXU4 = X,local [\"EXU5\",\"EXU6\"] [EXU5 = EXU4,EXU6 = nil(),EXU3 = '|'(1:EXU5 2:EXU6)]],\"Append\" \"EXU2\" \"EXU3\" \"EXU1\"]] else [skip]]]local [\"EXU1\"] [EXU1 = L1,\"Reverse\" \"EXU1\" \"Out1\"]skip/BOut1skip/f"

Out1 : [3 2 1]

Store : ((68, 70, 66, 42), '|'(1:61 2:62)),

((69, 55), nil()),

((67, 54, 53, 32, 16), 2),

((65, 57, 59, 45), '|'(1:54 2:55)),

((63, 56, 51, 50, 36, 18), 3),

((64), '|'(1:67 2:68)),

((61, 60, 28, 14), 1),

((62), nil()),

((58, 52), nil()),

((44, 48), '|'(1:51 2:52)),

((49, 38, 19), nil()),

((47), nil()),

((46, 34, 17), '|'(1:18 2:19)),

((43, 30, 15), '|'(1:16 2:17)),

((41), '|'(1:56 2:57)),

((40, 26, 9), '|'(1:14 2:15)),

((37, 39, 35, 31, 27, 10), '|'(1:20 2:21)),

((33), '|'(1:36 2:37)),

((29), '|'(1:32 2:33)),

((24), 6),

((25), nil()),

((22), 5),

((23), '|'(1:24 2:25)),

((20), 4),

((21), '|'(1:22 2:23)),

((8), proc(["Ls","Ms","EXU1"],[case Ls of nil() then [EXU1 = Ms] else [case Ls of '|'(1:X 2:Lr) then [local ["EXU2","EXU3"] [EXU2 = X,local ["EXU4","EXU5"] [EXU4 = Lr,EXU5 = Ms,"Append" "EXU4" "EXU5" "EXU3"],EXU1 = '|'(1:EXU2 2:EXU3)]] else [skip]]],[("Append",8)])),

((11), '|'(1:28 2:29)),

((12), proc(["Xs","EXU1"],[case Xs of nil() then [EXU1 = nil()] else [case Xs of '|'(1:X 2:Xr) then [local ["EXU2","EXU3"] [local ["EXU4"] [EXU4 = Xr,"Reverse" "EXU4" "EXU2"],local ["EXU4"] [EXU4 = X,local ["EXU5","EXU6"] [EXU5 = EXU4,EXU6 = nil(),EXU3 = '|'(1:EXU5 2:EXU6)]],"Append" "EXU2" "EXU3" "EXU1"]] else [skip]]],[("Reverse",12),("Append",8)])),

((13), '|'(1:63 2:64)),

((1), Primitive Operation),

((2), Primitive Operation),

((3), Primitive Operation),

((4), Primitive Operation),

((5), Primitive Operation),

((6), Primitive Operation),

((7), Primitive Operation)

Mutable Store: Empty

Current Environment : ("Append" -> 8, "L1" -> 9, "L2" -> 10, "Out" -> 11, "Reverse" -> 12, "Out1" -> 13, "IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4, "LT" -> 5, "Mod" -> 6, "IntMultiply" -> 7)

Stack : ""

```
   Out1 = {Reverse L1}
   skip Browse Out1
   skip Full
end
```

[Explanation:

L1 and L2 are originally set to the lists [1, 2, 3] and [4, 5, 6], respectively.

To combine two lists, we use the Append function. It mixes the arguments L1 and L2 given to it.

It examines the L1 structure within the Append function:

It initially notices 1 and the last item on list Lr, which is [2, 3].

It adds 1 to the outcome.

Then, using Lr and L2, it calls itself in a loop.

Until L1 is processed completely, this process continues. The result is [1, 2, 3, 4, 5, 6].

A list is reversed using the Reverse function. It uses L1 as an argument and reverses the elements.

The store serves as a record of data changes made while a program is running. Values and their locations in memory are tracked by it.

The environment displays the bindings of variables and functions as they are right now.

The functions Append and Reverse have been compiled and are stored at certain locations in memory (for example, Append is stored at position 8), as can be seen from the store and environment information.

n simple terms, the code manipulates lists L1 and L2 by appending them together to create Out and reversing L1 to create Out1

B. ----------Append_diff.txt----------

```
// Append example with difference lists
local L1 End1 L2 End2 H1 T1 H2 T2 LNew in
   L1 = ((1|(2|End1)) # End1)       // List [1,2] as a difference list
   L2 = ((3|(4|End2)) # End2)       // List [3,4] as a difference list

   L1 = (H1 # T1)                   // Pattern match, name head and tail
   L2 = (H2 # T2)                   // Pattern match, name head and tail
   T1 = H2                         // Bind/unify tail of L1 with head of L2

   LNew = (L1 # T2)                 // Build a new difference list

   skip Browse LNew
   skip Full
end

// Testing iterative Reverse function
local Reverse L1 Out1 in

   // O(n) version of Reverse on p 148 (modified for hoz):
   fun {Reverse Xs} Y1 ReverseD in
      proc {ReverseD Xs Y1 Y}
         case Xs
         of nil then Y1 = Y
         [] '|'(1:X 2:Xr) then {ReverseD Xr Y1 (X|Y)}
         end
      end
      {ReverseD Xs Y1 nil}
      Y1
```

```
   end

   L1 = (1|(2|(3|(4|nil))))
   Out1 = {Reverse L1}
   skip Browse Out1
   skip Full
end
```

[OUTPUT]

```
ghci> runFull "declarative" "append_diff.txt" "append_diff2kern.txt"
LNew : '#'(1:35 2:36)

Store : ((36, 24, 28, 11, 33, 15), Unbound),
((35, 8, 31), '#'(1:17 2:18)),
((18, 22, 9, 30, 13, 23, 32, 14), '|'(1:25 2:26)),
((10, 34), '#'(1:23 2:24)),
((17, 29, 12), '|'(1:19 2:20)),
((27), 4),
((25), 3),
((26), '|'(1:27 2:28)),
((21), 2),
((19), 1),
((20), '|'(1:21 2:22)),
((16), '#'(1:35 2:36)),
((1), Primitive Operation),
((2), Primitive Operation),
((3), Primitive Operation),
```

((4), Primitive Operation),

((5), Primitive Operation),

((6), Primitive Operation),

((7), Primitive Operation)

Mutable Store: Empty

Current Environment : ("L1" -> 8, "End1" -> 9, "L2" -> 10, "End2" -> 11, "H1" -> 12, "T1" -> 13, "H2" -> 14, "T2" -> 15, "LNew" -> 16, "IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4, "LT" -> 5, "Mod" -> 6, "IntMultiply" -> 7)

Stack : "local [\"Reverse\",\"L1\",\"Out1\"] [Reverse = proc {$ Xs EXU1} [local [\"Y1\",\"ReverseD\"] [ReverseD = proc {$ Xs Y1 Y} [case Xs of nil() then [Y1 = Y] else [case Xs of '|'(1:X 2:Xr) then [local [\"EXU2\",\"EXU3\",\"EXU4\"] [EXU2 = Xr,EXU3 = Y1,local [\"EXU5\",\"EXU6\"] [EXU5 = X,EXU6 = Y,EXU4 = '|'(1:EXU5 2:EXU6)],\"ReverseD\" \"EXU2\" \"EXU3\" \"EXU4\"]] else [skip]]],local [\"EXU2\",\"EXU3\",\"EXU4\"] [EXU2 = Xs,EXU3 = Y1,EXU4 = nil(),\"ReverseD\" \"EXU2\" \"EXU3\" \"EXU4\"],EXU1 = Y1]],local [\"EXU1\",\"EXU2\"] [EXU1 = 1,local [\"EXU3\",\"EXU4\"] [EXU3 = 2,local [\"EXU5\",\"EXU6\"] [EXU5 = 3,local [\"EXU7\",\"EXU8\"] [EXU7 = 4,EXU8 = nil(),EXU6 = '|'(1:EXU7 2:EXU8)],EXU4 = '|'(1:EXU5 2:EXU6)],EXU2 = '|'(1:EXU3 2:EXU4)],L1 = '|'(1:EXU1 2:EXU2)],local [\"EXU1\"] [EXU1 = L1,\"Reverse\" \"EXU1\" \"Out1\"],skip/BOut1,skip/f]"

Out1 : [4 3 2 1]

Store : ((39, 70, 65, 60, 55, 52, 49, 71), '|'(1:72 2:73)),

((73, 66), '|'(1:67 2:68)),

((72, 46), 4),

((69, 47), nil()),

((68, 61), '|'(1:62 2:63)),

((67, 44), 3),

((64, 45), '|'(1:46 2:47)),

((63, 56), '|'(1:57 2:58)),

((62, 42), 2),

((59, 43), '|'(1:44 2:45)),

((58, 53), nil()),

((57, 40), 1),

((54, 41), '|'(1:42 2:43)),

((51, 48, 38), '|'(1:40 2:41)),

((50), proc(["Xs","Y1","Y"],[case Xs of nil() then [Y1 = Y] else [case Xs of '|'(1:X 2:Xr) then [local ["EXU2","EXU3","EXU4"] [EXU2 = Xr,EXU3 = Y1,local ["EXU5","EXU6"] [EXU5 = X,EXU6 = Y,EXU4 = '|'(1:EXU5 2:EXU6)],"ReverseD" "EXU2" "EXU3" "EXU4"]] else [skip]]],[("ReverseD",50)])),

((37), proc(["Xs","EXU1"],[local ["Y1","ReverseD"] [ReverseD = proc {$ Xs Y1 Y} [case Xs of nil() then [Y1 = Y] else [case Xs of '|'(1:X 2:Xr) then [local ["EXU2","EXU3","EXU4"] [EXU2 = Xr,EXU3 = Y1,local ["EXU5","EXU6"] [EXU5 = X,EXU6 = Y,EXU4 = '|'(1:EXU5 2:EXU6)],"ReverseD" "EXU2" "EXU3" "EXU4"]] else [skip]]],local ["EXU2","EXU3","EXU4"] [EXU2 = Xs,EXU3 = Y1,EXU4 = nil(),"ReverseD" "EXU2" "EXU3" "EXU4"],EXU1 = Y1]],[])),

((36, 24, 28, 11, 33, 15), Unbound),

((35, 8, 31), '#'(1:17 2:18)),

((18, 22, 9, 30, 13, 23, 32, 14), '|'(1:25 2:26)),

((10, 34), '#'(1:23 2:24)),

((17, 29, 12), '|'(1:19 2:20)),

((27), 4),

((25), 3),

((26), '|'(1:27 2:28)),

((21), 2),

((19), 1),

((20), '|'(1:21 2:22)),

((16), '#'(1:35 2:36)),

((1), Primitive Operation),

((2), Primitive Operation),

((3), Primitive Operation),

((4), Primitive Operation),

((5), Primitive Operation),

((6), Primitive Operation),

((7), Primitive Operation)

Mutable Store: Empty

Current Environment : ("Reverse" -> 37, "L1" -> 38, "Out1" -> 39, "IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4, "LT" -> 5, "Mod" -> 6, "IntMultiply" -> 7)

Stack : ""

[Explanation: Two operations are carried out: the reversal of a list (L1) to get Out1, and the creation of a new difference list (LNew) by concatenating two difference lists (L1 and L2).
L1 and L2 are implemented as difference lists. A list where the head and tail are stored separately is represented as a difference list. It makes concatenation effective.

Difference List Concatenation (LNew):

A difference list with a head (H1) and a tail (T1) is known as L1.

A difference list with a head (H2) and a tail (T2) is known as L2.

The value of H2 is allocated to T1, connecting L1's tail to L2's head.

L1 and T2 are combined to become LNew. The effect of merging L1 and L2 together is represented by the new difference list, LNew, that is produced by this concatenation.

List reversed (Out1):

A list can be reversed using the Reverse function.

An accumulator Y1 is utilized inside the Reverse method's helper function ReverseD to reverse the list. It repeatedly flips the list, and Y1 is given the outcome.

The list [1, 2, 3, 4] is the definition of L1.

By using the Reverse function on L1, the elements are essentially reversed to produce Out1.

C. ----------Combined----------

List = (1(2(3(4(5(6nil))))))

Append { Reverse{(2|(3|(4|(5|(6|nil))))) (1|nil)}

Append { Reverse{(3|(4|(5|(6|nil)))) (2|nil)}

Append { Reverse{(4|(5|(6|nil))) (3|nil)}

Append { Reverse{(5|(6|nil))) (4|nil)}

Append { Reverse{(6|nil)) (5|nil)}

Append { Reverse{(nil) (6|nil)}

// We combine them now //

Append{nil (6|nil)}

Append{(nil|6) {5|nil)}

Append{(nil|(6|(5|nil))) {4|nil)}

Append{((nil|(6|(5|(4|nil)))) {3|nil)}

Append{((nil|(6|(5|(4|(3|nil))))) {2|nil)}

Append{((nil|(6|(5|(4|(3|(2|nil)))))) {1|nil)}

Append{((nil|(6|(5|(4|(3|(2|(1|nil))))))) {nil)}

Output: 6 5 4 3 2 1

[Explanation: Recursively reversing a list

It defines a "Reverse" function that accepts a list of "Xs" as input.

A helper function called "ReverseD" inside of "Reverse" actually reverses the direction.

Recursively going through the input list, "ReverseD" creates a new list in reverse order.

The '|' operator is used to append each processed element to the result list.

Up until the entire list is reversed, the recursion keeps going.

It reverses a list of length 6 using various cons operations throughout the recursion, totaling 15 cons operators.]

{ReverseD (2|(3|(4|(5|(6|nil))))) Y1 '|'(1:1 2:Y)}

{ReverseD (3|(4|(5|(6|nil))))) Y1 '|'(1:2 2:'|'(1:1 2:Y))}

{ReverseD (4|(5|(6|nil))) Y1 '|'(1:3 2:'|'(1:2 2: '|'(1:1 2:Y)))}

{ReverseD (5|(6|nil)) Y1 '|'(1:4 2:'|'(1:3 2:'|'(1:2 2: '|'(1:1 2:Y))))}

{ReverseD (6|nil) Y1 '|'(1:5 2:'|'(1:4 2:'|'(1:3 2:'|'(1:2 2: '|'(1:1 2:Y)))))}

{ReverseD (nil) Y1 '|'(1:6 2: '|'(1:5 2:'|'(1:4 2:'|'(1:3 2:'|'(1:2 2: '|'(1:1 2:Y)))))) }

Y '|'(1:6 2: '|'(1:5 2:'|'(1:4 2:'|'(1:3 2:'|'(1:2 2: '|'(1:1 2:Y)))))) }

'|'(1:6 2: '|'(1:5 2:'|'(1:4 2:'|'(1:3 2:'|'(1:2 2: '|'(1:1 2:nil))))))

Output:- [6 5 4 3 2 1]

[Explanation: An alternative approach to reversing the list using an accumulator variable "Y." It defines a "Reverse" function that accepts a list of "Xs" as input. An accumulator "Y" is used by the helper function "ReverseD" inside "Reverse" to store the reversed list. Instead of using recursive calls to traverse the input list, "ReverseD" simply prepends elements to the accumulator "Y". This method lowers cons operations and reduces the necessity for nested recursive calls. For reversing a list of length 6, it utilizes only 6 cons operators.]