Eric Smrkovsky
CSCI 117
9/21/2019

Part 1

my_s2k1.txt

```
local A in
  A = false()
  local B in
    local T in
      T = true()
      if T then
        skip Browse A
      else
        if B then
          skip Basic
        else
          skip Basic
        end
      end
    end

  case A of
    tree() then
      skip Basic
    else
      case A of
        false() then
          skip Basic
      else
        case A of
          true() then
            skip Basic
        else
          skip Basic
        end
      end
    end
  end
end
```

1a

If

The first half of this program uses nested if statements to check a boolean condition and then browses it. The first nested if statement checks a condition that doesn't even have a verifiable condition to check.

All the ends are in one line.

Uses UniqueIF for new variables.

Created an unnecessary local statement in the first else creating a variable to check the conditional with rather than just using the B variable. This could be because B is unbound?

Case

The code for the nested case statements practically matched my translation except that it is staggered differently, and all of the ends are in rows.

It seems that within **if** statements the kernel syntax wants to create a new variable and assign it the variable that is outside of its scope, but the **case** statements don't mind using them.

Eric Smrkovsky
CSCI 117
9/21/2019

my_s2k2.txt

```
1  local A in
2    A = 2
3    local Check in
4      local Aye in
5        local One in
6          One = 1
7          Aye = A
8          {Eq One Aye Check}
9        end
10     end
11   if Check then
12     skip Basic
13   else
14     skip Basic
15     end
16   end
17   local Check2 in
18     local Aye2 in
19       local Three in
20         local One in
21           local Sub in
22             Aye2 = A
23             Three = 3
24             One = 1
25             {IntMinus Three One Sub}
26             {Eq Aye2 Sub Check2}
27           end
28         end
29       end
30     end
31     if Check2 then
32       skip Browse A
33     else
34       skip Basic
35       end
36     end
37   end
```

1b

In the second part of sugar.txt we find **expressions** being used to check the condition of two **if statements**. It is surprising the amount of extra work necessary to translate from the syntactic sugar to regular kernel code. For each **expression** the parser must create variables and assign them values so that they can be used within procedures. The procedures are then used to assign new values to the variables that are used within the condition checking part of the **if statements**. Only then will the **if statement** be able to run.

A difference I noticed between my version and the one that is produced in sugar2kern.txt is that line 22 and 26 from my code are placed in a different scope of a local statement. Maybe that is because the parser noticed that the variables that are being bound with those two lines of code aren't used in the scope of the local statements that I wrote them in.

Eric Smrkovsky
CSCI 117
9/21/2019

my_s2k3.txt

```
local T in
  local Three in
    Three = 3
    T = tree(1:Three 2:T)
  end
  local X in
    local Y in
      local A in
        local B in
          local Tree in
            Tree = tree(1:A 2:B)
            Tree = T
          end
        end
      end
    end
  end
local Check in
  local One in
    local Onee in
      One = 1
      Onee = 1
      {Eq One Onee Check}
    end
  end
  if Check then
    local B in
      local Five in
        local Two in
          Five = 5
          Two = 2
          {IntMinus Five Two B}
        end
      end
      local Z in
        skip Browse B
      end
    end
  else
    skip Basic
    end
  end
end
```

1c

In the third part of sugar.txt we see to examples of declaring variables without doing it explicitly with local statements. Here we see all the variables are declared in a pattern and then that pattern is bound to a variable. As you can see in my translation into the kernel code, it takes something that would normally take many steps and does it in just a few.

This is known as “implicit variable initialization.”

Also, the line after the if statement is known as an “in statement,” it is a way to declare variables without using the term “local.”

Eric Smrkovsky
CSCI 117
9/21/2019

my_s2k4.txt

```
local Fun in
  local R in
    Fun = proc {$ X Out}
      Out = X
      end
    local Four in
      Four = 4
      {Fun Four R}
    skip Browse R
    end
  end
end
```

1d

In part four of sugar.txt I can see a function being defined and used. The biggest thing that I see between my translation and the one given is that I use a procedure in place of a function. I procedure requires defining the output while a function just returns the last thing in its body. The program then calls the function and binds it’s return value to R. Then the program displays R.

Eric Smrkovsky
CSCI 117
9/21/2019

my_s2k5.txt

```
local A in
  local B in
    skip Basic
    local List in
      local Four in
        Four = 4
        local Pair in
          Pair = '#'(1:B 2:B)
          List = rdc(1:Four 2:B 3:Pair)
        end
      end
    A = List
    end
    local Five in
      Five = 5
      local Sub in
        local Three in
          local Four in
            Three = 3
            Four = 4
            {IntMinus Four Three Sub}
          end
        end
      {IntPlus Five Sub B}
      end
    end
    skip Browse A
    skip Browse B
    skip Store
  end
end
```

1e

Here is another example of how many underlying steps are happening behind the scenes of a program that uses syntactic sugar.

For the variable A to be bound to the record rdc first the program must create variables and store locations for the arity. The first feature is assigned the value 4 so a variable is assigned and bound for it, next the second feature is bound to the store location that was created for B, and then a record is produced for the last feature. All of this is done within the binding to A using syntactic sugar.

Eric Smrkovsky
CSCI 117
9/21/2019

Part 2A

```
// Append function p 133

local Append L1 L2 Out Reverse L3 Out1 in

  Append = fun {$ Ls Ms}
    case Ls
    of nil then Ms
    [] '|'(1:X 2:Lr) then Y in
      Y = {Append Lr Ms}
      //skip Full
      (X|Y)
    end
  end

// Reverse function p 135 in ct
  Reverse = fun {$ Xs}
    case Xs
    of nil then nil
    [] '|'(1:X 2:Xr) then               //case stmt (head tail)
      {Append {Reverse Xr} (X|nil)}  //outputs the "head" of reversed list
    end
  end

  L3 = (7|(8|(9|(10|nil))))     //a list of size 4
  Out1 = {Reverse L3}           //now use it in function

  skip Full                     //display Store/Env/Stack
  skip Browse Out1              //show me the "head" of the reversed list
end
```

Eric Smrkovsky
CSCI 117
9/21/2019

Store/Environment/Description

```
Current Environment : ("Out1" -> 11,
 -> 8, "L2" -> 7, "L1" -> 6, "Append"
 -> 2, "Eq" -> 3, "GT" -> 4)
Stack : "skip/BOut1"

Out1 : '|'(1:19 2:47)
```

```
Store : ((53, 55, 52, 49, 23), '|'(1:13 2:46)),
((54, 39), nil()),
((51, 43, 45, 42, 26), '|'(1:15 2:39)),
((50), '|'(1:15 2:53)),
((48, 40), '|'(1:17 2:43)),
((47), '|'(1:17 2:50)),
((46), nil()),
((44, 35), nil()),
((41, 36, 38, 29), '|'(1:17 2:35)),
((37, 34), nil()),
((28, 32), '|'(1:19 2:34)),
((33, 20), nil()),
((31), nil()),
((30, 18), '|'(1:19 2:20)),
((27, 16), '|'(1:17 2:18)),
((25), '|'(1:19 2:36)),
((24, 14), '|'(1:15 2:16)),
((22), '|'(1:19 2:40)),
((21, 10, 12), '|'(1:13 2:14)),
((19), 10),
((17), 9),
((15), 8),
((13), 7),
((11), '|'(1:19 2:47)),
```

Here I have displayed the relevant environment, store and the output of the reverse function. In the current environment you can see that "Out1" has been assigned the store location of "11." Within in the store you can see that the location "11" has been assigned a record which contains two store locations as it's features. This is where we will start with determining the output. When we reach a "nil" value we are done.

((11), '|'(1:19 2:47)) => ((19), 10) => 10

((47), '|'(1:17 2:50)) => ((17), 9) => 9

((50), '|'(1:15 2:53)) => ((15), 8) => 8

((53), '|'(1:13 2:46)) => ((13), 7) => 7

((46), nil()) =>END

So, as you can see the output is 10 9 8 7.

Eric Smrkovsky
CSCI 117
9/21/2019

Part2B)

Reverse2.txt

```
local Dl Reverse Out in
  Dl = (1|(2|(3|(4|nil))))      // a d

// Reverse function p 148
  Reverse = fun {$ Xs}
    local ReverseD Y1 N in
      ReverseD = proc {$ Xs Y1 Y}
        case Xs
          of nil then Y1=Y
          [] '|'(1:X 2:Xr) then L in
            L = '|'(1:X 2:Y)
            {ReverseD Xr Y1 L}
        end
      end
    N = nil
    {ReverseD Xs Y1 N}
    skip Browse Y1
    Y1
    end
  end
  Out = {Reverse Dl}
  skip Full
  skip Browse Out
end
```

Eric Smrkovsky
CSCI 117
9/21/2019

Store/Environment/Description

```
Store : ((7, 19, 24), '|'(1:15 2:23)),
((23), '|'(1:13 2:22)),
((22), '|'(1:11 2:21)),
((21), '|'(1:9 2:20)),
((20), nil()),
((18), proc),
((17, 5, 8), '|'(1:9 2:10)),
((16), nil()),
((15), 4),
((14), '|'(1:15 2:16)),
((13), 3),
((12), '|'(1:13 2:14)),
((11), 2),
((10), '|'(1:11 2:12)),
((9), 1),
((6), proc),
((1), Primitive Operation),
((2), Primitive Operation),
((3), Primitive Operation),
((4), Primitive Operation)

Current Environment : ("Out" -> 7, "Rev
" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT
Stack : "skip/BOut"

Out : '|'(1:15 2:23)
```

Here it is pretty much the same story as the output of 2a except that less store locations are created. The output record "Out" is assigned the store location "7," whose value is a record just like 2a, ends when a nil is reached. Here is how to derive the results.

((7), '|'(1:15 2:23)) => ((15), 4) => 4

((23), '|'(1:13 2:22)) => ((13), 3) => 3

((22), '|'(1:11 2:21)) => ((11), 2) => 2

((21), '|'(1:9 2:20)) => ((9), 1) => 1

((20), nil()) => DONE

So, the output is 4 3 2 1.

Eric Smrkovsky
CSCI 117
9/21/2019

Part2C

In the first reverse function (a), within the body the append function is used recursively and uses the reverse function as input. So, the first reverse function uses a total of 24 Cons operations to construct the output list and is quadratic. The second reverse (b) function uses a helper function and is linear, this uses 6 Cons operations to construct the output list.

In (a) there is two functions that are preforming cons operations so that is why it is quadratic, in (b) it appends within the function, so it does the cons operation linearly.