Eric Smrkovsky
CSci 117
Lab 14
11/23/19

P1Q1a) I am going to start my explanation at the first try statement. When the first try statement begins execution the (catch divByZero(…) then Browse X end) statement is pushed onto the stack. After that the nested try begins execution the (catch illFormedExpr(E) then … end) statement is pushed onto the stack. After this is done, then {Browse {Eval times(6 11)}} is pushed followed by {Browse {Eval plus(plus(5 5) 10)}} and finally {Browse {Eval minus(7 10)}} is pushed onto the stack and begins execution. Next the Eval minus function begins execution pushing all of it's statements onto the stack as well.

Continuing execution within the Eval function the first if fails, causing the else to execute. The else contains a Case statement and none of it's statements matches with the function Minus so the final else is executed. Within this else the Raise statement is executed and starts to pop everything off of the stack until it reaches it's marker (illFormedExpr(E)). Now finally the nested catch statement is popped from the stack leaving {Browse´*** Illegal expression´#E#´ ***´} as the next statement in the stack. The program executes the Browse and the program is finished displaying an error.

P1Q1b) I am going to start my explanation at the first try statement. When the first try statement begins execution the (catch divByZero(…) then Browse X end) statement is pushed onto the stack. After that the nested try begins execution the (catch illFormedExpr(E) then … end) statement is pushed onto the stack. After this is done, then {Browse {Eval times(6 11)}} is pushed followed by {Browse {Eval divide(6 0)}} and finally {Browse {Eval plus(plus(5 5) 10)}} is pushed onto the stack and begins execution. Next the Eval plus(…) function begins execution pushing all of it's statements onto the stack and executes {Browse {Eval plus(plus(5 5) 10)}} without raising any errors. The first Eval function only executes as far as the first part of the case statement.

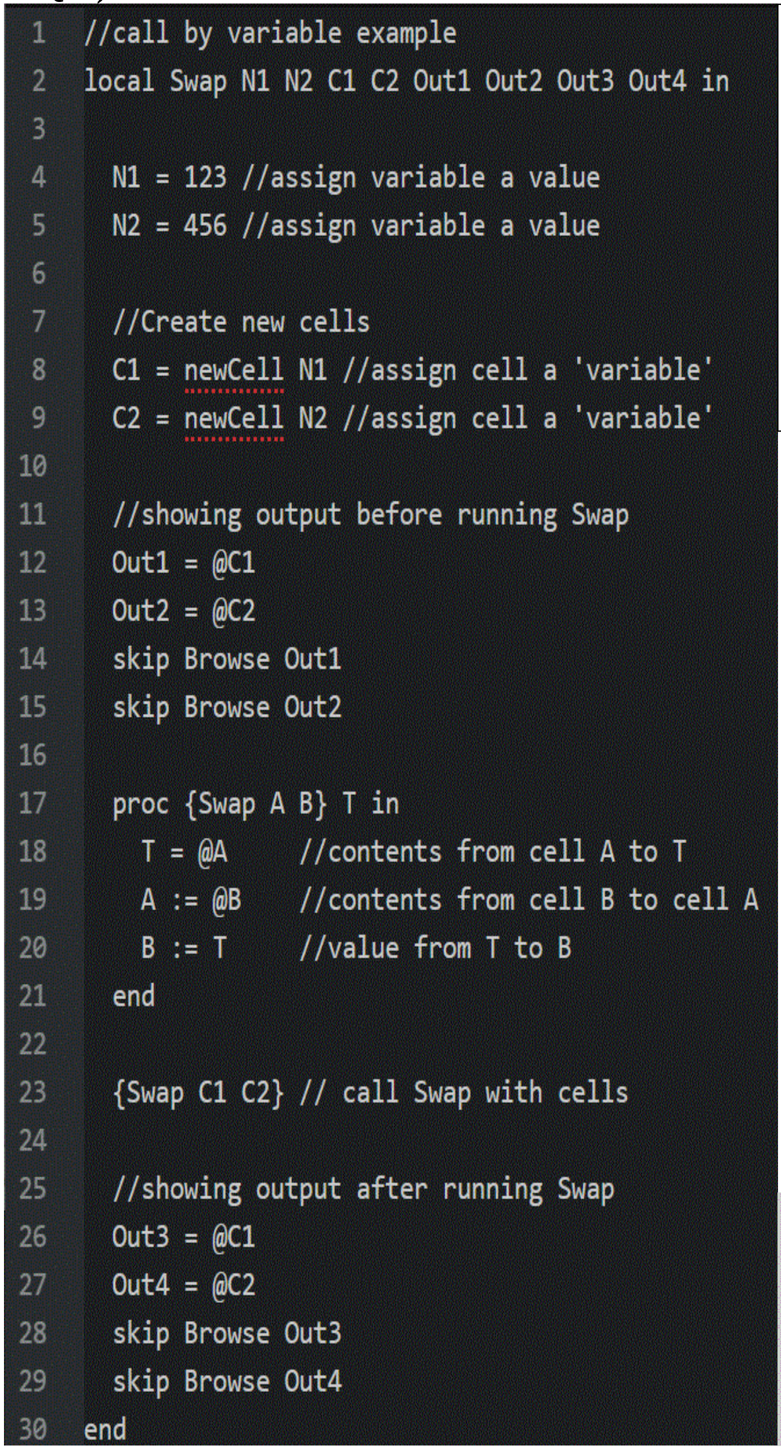
(Eval plus(plus(5 5) 10) executes with a recursive call for plus(5 5) which results in plus(10 10) being browsed displaying 20) (total of 4 recursive calls to Eval, 3 stopping at IsNumber)

Next, {Browse {Eval divide(6 0)}} is pushed onto the stack. Execution starts again with Eval, this time it reaches the divide part of the case statement. (if {Eval Y} == 0) is true so the Raise divByZero statement is executed. Now all of the statements on the stack are popped until a catch statement that matches divByZero(E) is found. {Browse {Eval times(6 11)}} is popped and then the (not nested) catch statement is popped from the stack as well. Lastly, {Browse X} is popped from the stack and the program finishes displaying 6 on the screen.

P1Q2) Yes, it is possible. According to the textbook, when raise is executed the program pops elements off the stack looking for a catch statement, if the stack is emptied and no catch is found, then stop execution with the error message "Uncaught exception".

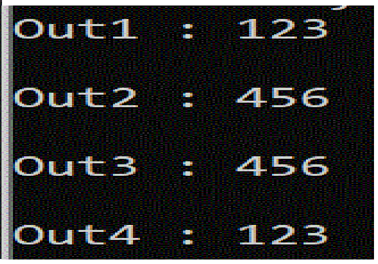
Eric Smrkovsky
CSci 117
Lab 14
11/23/19

P2Q1a)



```
//call by variable example
local Swap N1 N2 C1 C2 Out1 Out2 Out3 Out4 in

  N1 = 123 //assign variable a value
  N2 = 456 //assign variable a value

  //Create new cells
  C1 = newCell N1 //assign cell a 'variable'
  C2 = newCell N2 //assign cell a 'variable'

  //showing output before running Swap
  Out1 = @C1
  Out2 = @C2
  skip Browse Out1
  skip Browse Out2

  proc {Swap A B} T in
    T = @A     //contents from cell A to T
    A := @B    //contents from cell B to cell A
    B := T     //value from T to B
  end

  {Swap C1 C2} // call Swap with cells

  //showing output after running Swap
  Out3 = @C1
  Out4 = @C2
  skip Browse Out3
  skip Browse Out4
end
```

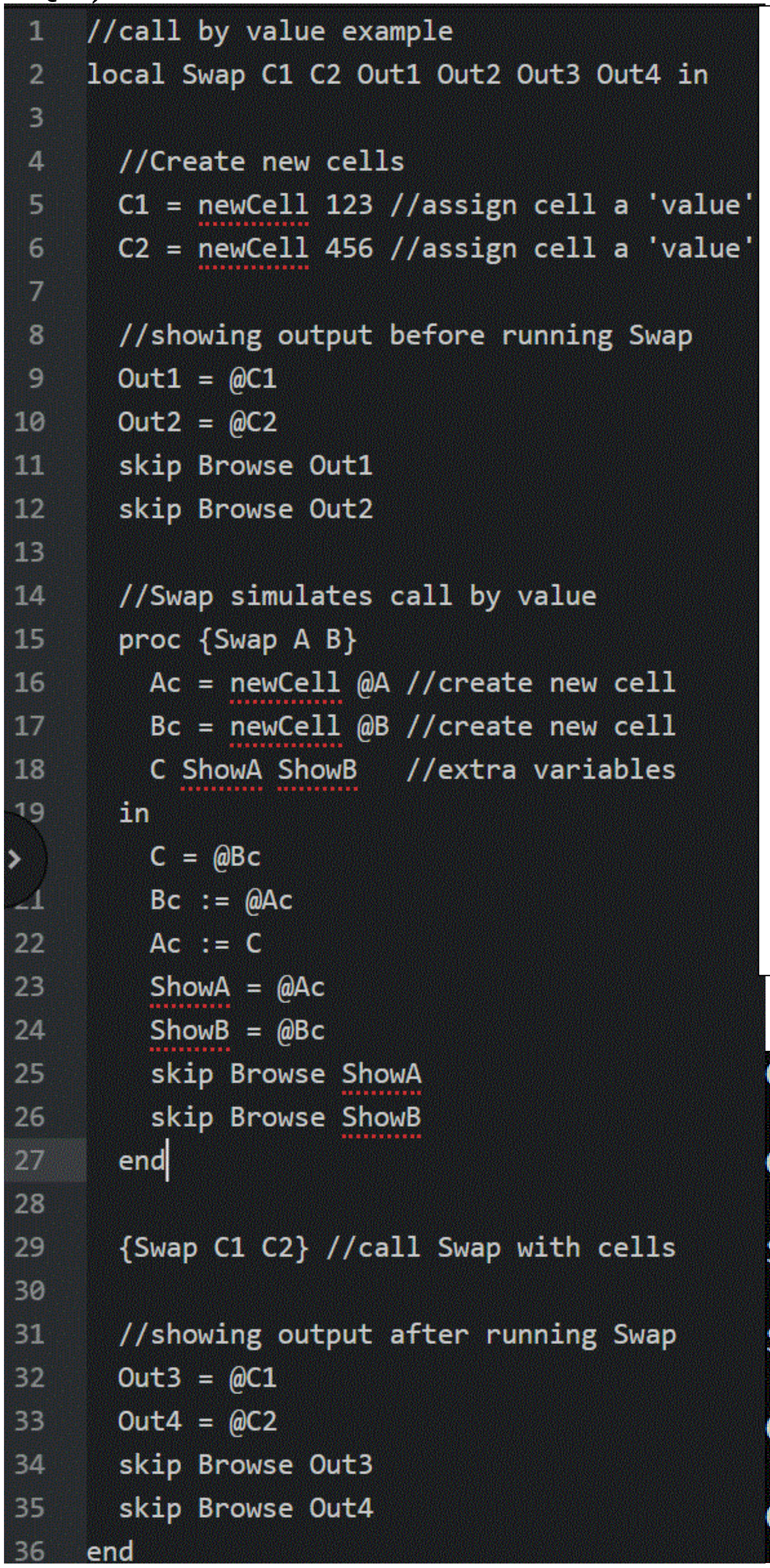
Here we can see that the variables N1 and N2 have been assigned values and the cells C1 and C2 are being used as a reference to N1 and N2. The references C1 and C2 are being passed into the swap function. The swap function takes the references, accesses the contents of the variable that they point to and swaps them. I will show the output of this function below. As you can see, the output shown after running swap have been reversed compared to before running the procedure.



```
Out1 : 123
Out2 : 456
Out3 : 456
Out4 : 123
```

Eric Smrkovsky
CSci 117
Lab 14
11/23/19

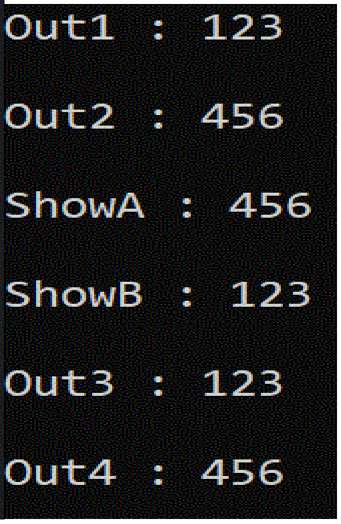
P2Q1b)



```
//call by value example
local Swap C1 C2 Out1 Out2 Out3 Out4 in

  //Create new cells
  C1 = newCell 123 //assign cell a 'value'
  C2 = newCell 456 //assign cell a 'value'

  //showing output before running Swap
  Out1 = @C1
  Out2 = @C2
  skip Browse Out1
  skip Browse Out2

  //Swap simulates call by value
  proc {Swap A B}
    Ac = newCell @A //create new cell
    Bc = newCell @B //create new cell
    C ShowA ShowB   //extra variables
  in
    C = @Bc
    Bc := @Ac
    Ac := C
    ShowA = @Ac
    ShowB = @Bc
    skip Browse ShowA
    skip Browse ShowB
  end

  {Swap C1 C2} //call Swap with cells

  //showing output after running Swap
  Out3 = @C1
  Out4 = @C2
  skip Browse Out3
  skip Browse Out4
end
```

In this program I will be simulating a call by value when I call the function swap. First I assign a value to each of the cells C1 and C2 and pass them into the function. Next, I create 2 new cells and assign the contents from the passed in cells to them. Then I access the values from those cells and swap them. This simulates passing by value because The new cells Ac and Bc have had their contents swapped locally within the function, but the variables that aren't local to the function are unchanged. This is hard to describe because I am trying to simulate passing by value. If I were passing a value into the function, the call would be {Swap 123 456}, I wouldn't be using cells. Unfortunately, I can't do that within the Oz syntax because the language uses a single assignment store, so we can only simulate it by using cells.

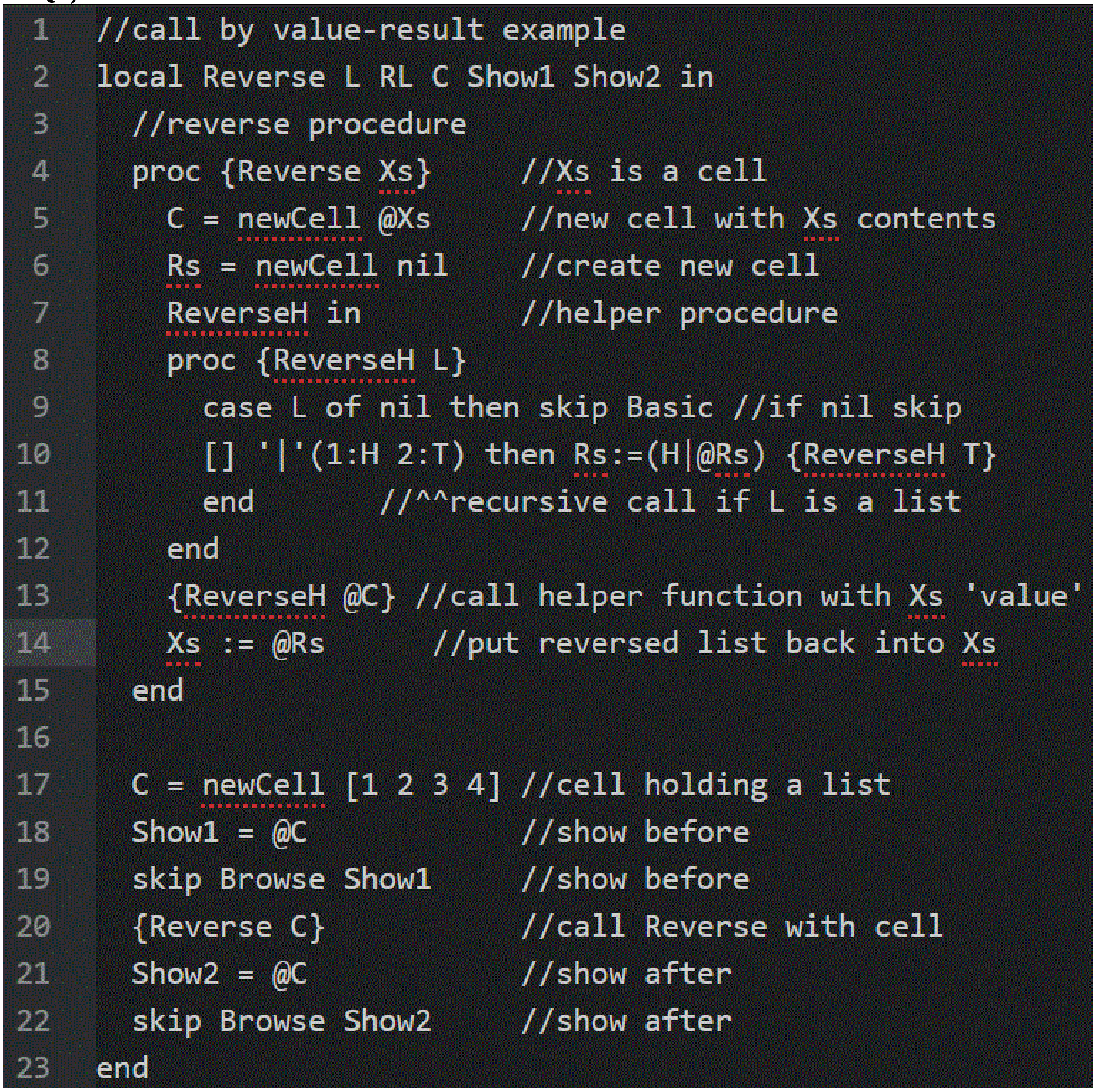
TRY RUNNING AS IN PIC



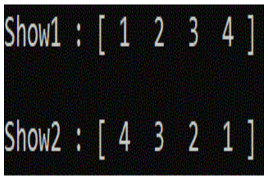
```
Out1 : 123

Out2 : 456

ShowA : 456

ShowB : 123

Out3 : 123

Out4 : 456
```

Eric Smrkovsky
CSci 117
Lab 14
11/23/19

P2Q2)



```
//call by value-result example
local Reverse L RL C Show1 Show2 in
  //reverse procedure
  proc {Reverse Xs}     //Xs is a cell
    C = newCell @Xs     //new cell with Xs contents
    Rs = newCell nil    //create new cell
    ReverseH in         //helper procedure
    proc {ReverseH L}
      case L of nil then skip Basic //if nil skip
      [] '|'(1:H 2:T) then Rs:=(H|@Rs) {ReverseH T}
      end       //^^recursive call if L is a list
    end
    {ReverseH @C} //call helper function with Xs 'value'
    Xs := @Rs      //put reversed list back into Xs
  end

  C = newCell [1 2 3 4] //cell holding a list
  Show1 = @C            //show before
  skip Browse Show1     //show before
  {Reverse C}           //call Reverse with cell
  Show2 = @C            //show after
  skip Browse Show2     //show after
end
```

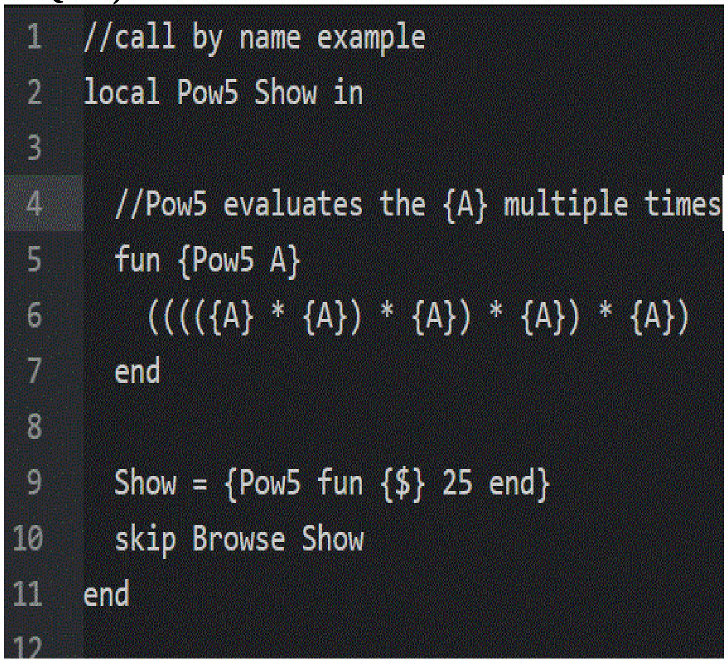


```
Show1 : [ 1  2  3  4 ]

Show2 : [ 4  3  2  1 ]
```

In this program I am showing an example of call by value-result. It is the same as call by value except in this case the cell is updated with the new reversed list before returning. What makes this call by value-result is how the data is accessed from within the reverse function. The cell with the list is passed into the function, but it is not used within Reverse's code, a new cell is used instead with Xs list as the contents. Afterwards Xs contents is updated with the reversed list so the original cell that was passed into Reverse is updated also when the procedure returns.

Eric Smrkovsky
CSci 117
Lab 14
11/23/19

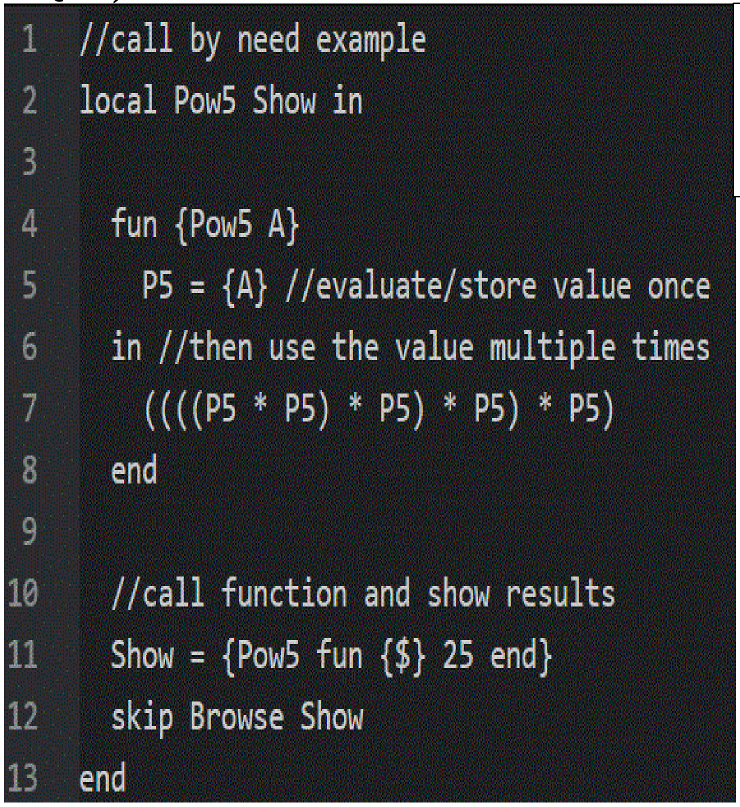
P2Q3a1)



```
//call by name example
local Pow5 Show in

  //Pow5 evaluates the {A} multiple times
  fun {Pow5 A}
    (((({A} * {A}) * {A}) * {A}) * {A})
  end

  Show = {Pow5 fun {$} 25 end}
  skip Browse Show
end
```

Here is an example of call by name. The function value is evaluated every time it is needed.

P2Q3a2)



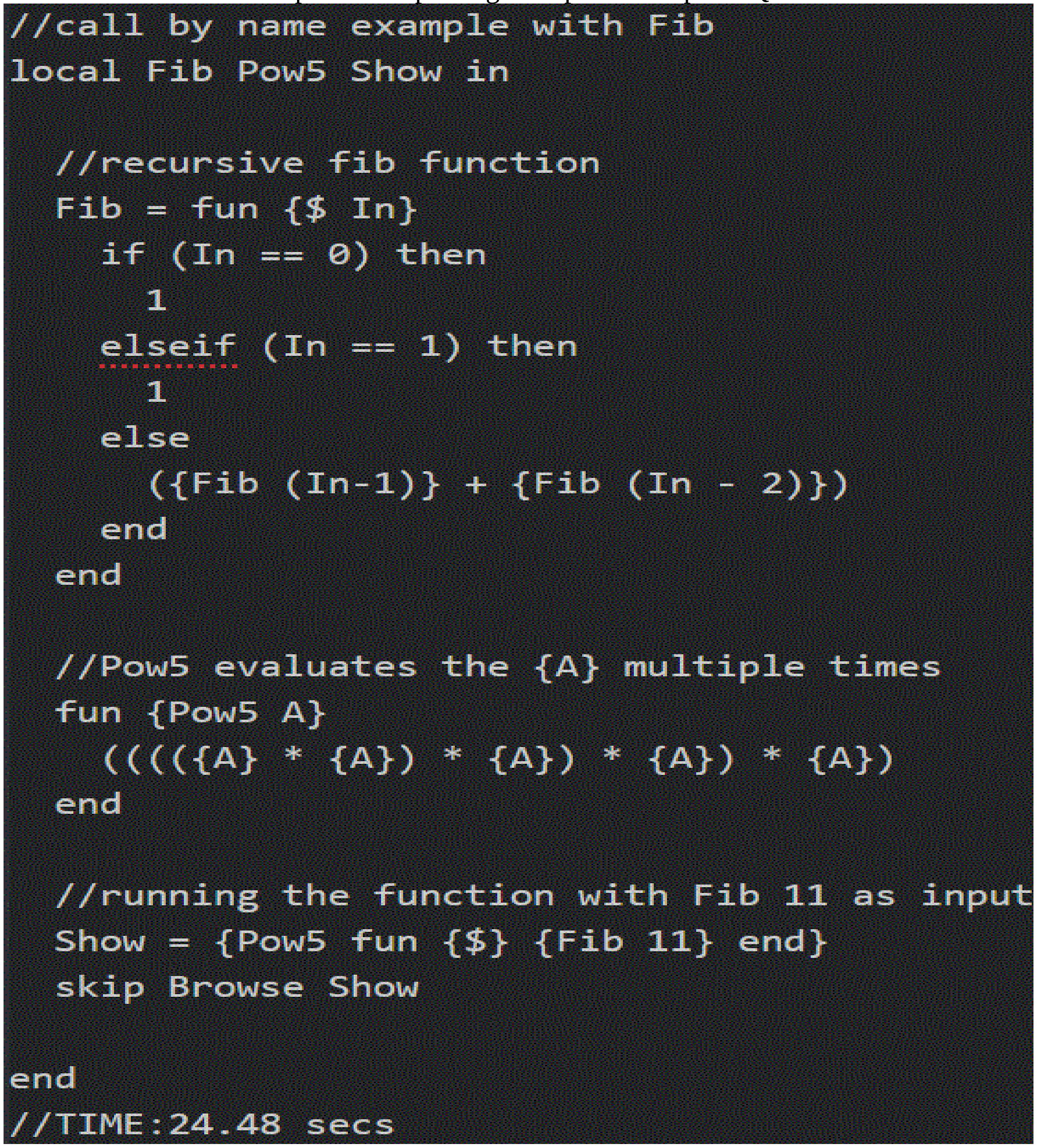
```
//call by need example
local Pow5 Show in

  fun {Pow5 A}
    P5 = {A} //evaluate/store value once
  in //then use the value multiple times
    ((((P5 * P5) * P5) * P5) * P5)
  end

  //call function and show results
  Show = {Pow5 fun {$} 25 end}
  skip Browse Show
end
```

Here is an example of call by need. The function value is only evaluated once. The value found during the evaluation is stored in a variable and used multiple times.

Eric Smrkovsky
CSci 117
Lab 14
11/23/19

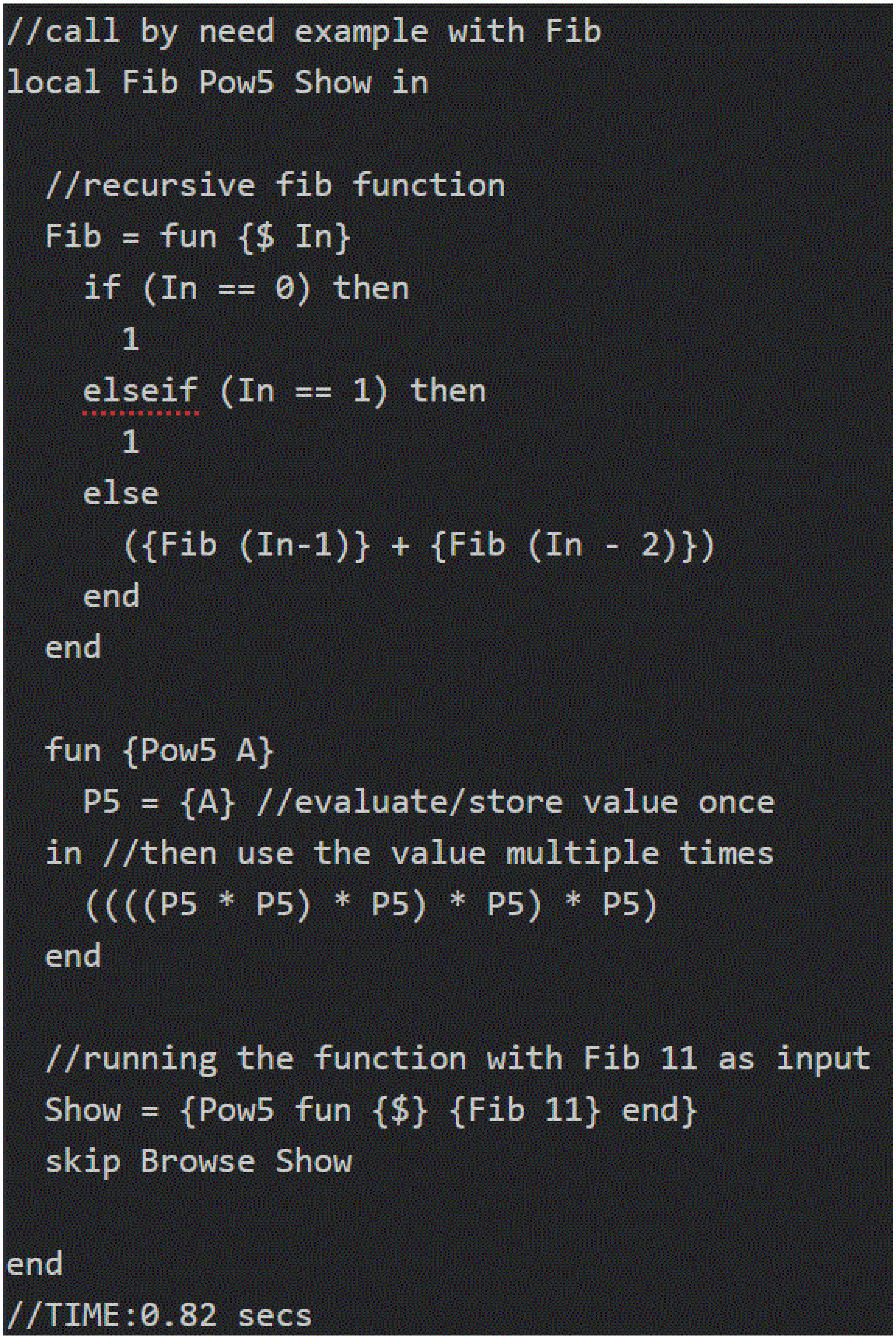
P2Q3b) Passing a parameter in the call by need method seems more efficient because the parameter (in these cases is a function) is memorized and then the memorized version is used multiple times. This is more efficient because the parameter is only evaluated once in the call by need method while it is evaluated every time it is needed in the call by name method.

P2Q3c) Below are two programs that use Fib as the input function for part Q3a, at the bottom of the code for each program I have included the execution time. As you can see the call by need version is 23.66 seconds faster then the call by name version. The reason for this difference is because of the method of parameter passing as explained in part 2Q3b above.



```
//call by name example with Fib
local Fib Pow5 Show in

  //recursive fib function
  Fib = fun {$ In}
    if (In == 0) then
      1
    elseif (In == 1) then
      1
    else
      ({Fib (In-1)} + {Fib (In - 2)})
    end
  end

  //Pow5 evaluates the {A} multiple times
  fun {Pow5 A}
    (((({A} * {A}) * {A}) * {A}) * {A})
  end

  //running the function with Fib 11 as input
  Show = {Pow5 fun {$} {Fib 11} end}
  skip Browse Show

end
//TIME:24.48 secs
```

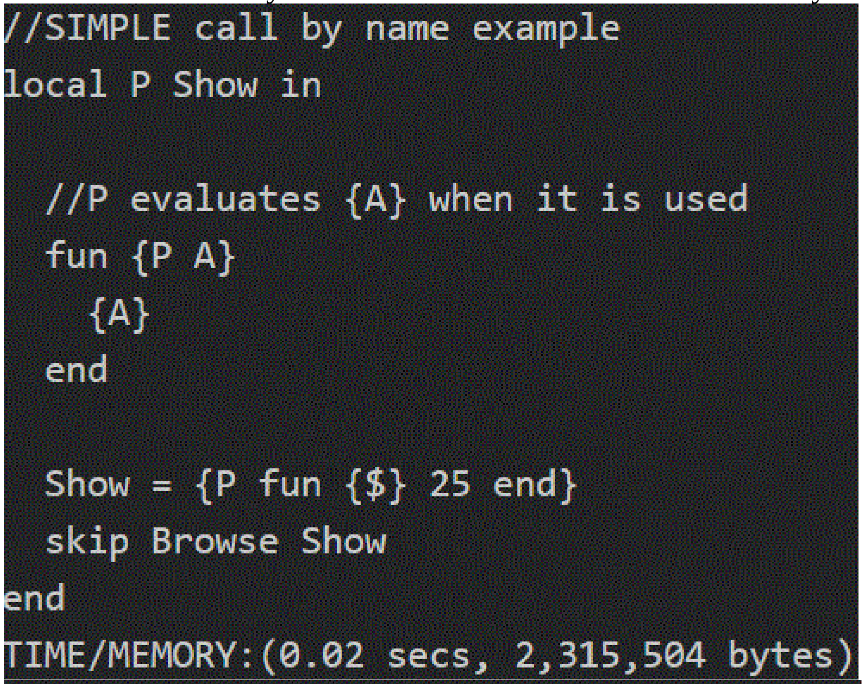
Eric Smrkovsky
CSci 117
Lab 14
11/23/19



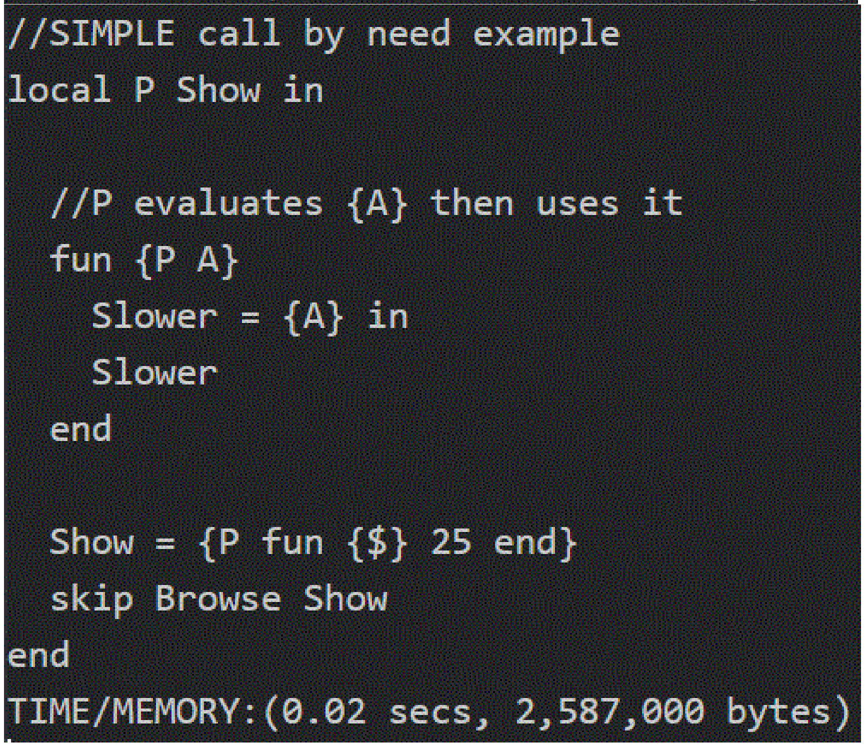
```
//call by need example with Fib
local Fib Pow5 Show in

  //recursive fib function
  Fib = fun {$ In}
    if (In == 0) then
      1
    elseif (In == 1) then
      1
    else
      ({Fib (In-1)} + {Fib (In - 2)})
    end
  end

  fun {Pow5 A}
    P5 = {A} //evaluate/store value once
  in //then use the value multiple times
    ((((P5 * P5) * P5) * P5) * P5)
  end

  //running the function with Fib 11 as input
  Show = {Pow5 fun {$} {Fib 11} end}
  skip Browse Show

end
//TIME:0.82 secs
```

Eric Smrkovsky
CSci 117
Lab 14
11/23/19

P2Q3d) Below are two programs that take a function as input, it is clear that the call by name version is more efficient because it uses 271,496 less bytes than the call by need version. The reason why this is more efficient is because the call by name program only needs to use the function once anyway. The call by need function has to evaluate the function before using it, so it costs more memory. Code is below with the time and memory consumption below it.



```
//SIMPLE call by name example
local P Show in

  //P evaluates {A} when it is used
  fun {P A}
    {A}
  end

  Show = {P fun {$} 25 end}
  skip Browse Show
end
TIME/MEMORY:(0.02 secs, 2,315,504 bytes)
```



```
//SIMPLE call by need example
local P Show in

  //P evaluates {A} then uses it
  fun {P A}
    Slower = {A} in
    Slower
  end

  Show = {P fun {$} 25 end}
  skip Browse Show
end
TIME/MEMORY:(0.02 secs, 2,587,000 bytes)
```

Eric Smrkovsky
CSci 117
Lab 14
11/23/19

P3Q1)
Program:

```
V := L
C := nil
while (@V \= nil)
  H|T = @V in
  C := H|@C
  V := T
End
```

Begin proof…
Invariant = P = {Rev(@V) ++ @C = Rev(L)}

Entering the loop:
(show the invariant is true for the initial values, when entering the loop)
@V = L, @C = nil = []
Rev(@V) ++ @C = Rev(L) ++ nil = Rev(L)   By substitution

Iteration:
(Assume the invariant is true at the beginning of the loop,
show the invariant is true at the end of the loop for the updated variables)
Assume : Rev(@V) ++ @C = Rev(L)
Let @V = H|T, @V' = T, @C' = H|@C

Rev(@V') ++ @C'	
= Rev(T) ++ H|@C	By substitution
= (Rev(T) ++ H) ++ @C	By associativity of append
= Rev(H|T) ++ @C	By definition of reverse
= Rev(@V) ++ @C	By substitution
= Rev(L)	By assumption

Exiting Loop:
(Assume the invariant is true and the loop condition is false, show that Out contains the desired value, L1 ++ L2)
Assume: Rev(@V) ++ @C = Rev(L), @V = nil
Rev(Nil) ++ @C = @C = Rev(L)

I have shown that the invariant is true before and after the loop. **…End proof**

Eric Smrkovsky
CSci 117
Lab 14
11/23/19

P3Q2)
Program:

```
V := L
C := 0
while (@V \= nil)
  H|T = @V in
  C := H + @C
  V := T
end
```

Begin proof…
P = {SumList(@V) + @C = SumList(L)}

Entering the loop:
(show the invariant is true for the initial values, when entering the loop)
@V = L, @C = 0
SumList(@V) + @C = SumList(L) + 0 = SumList(L)   By substitution

Iteration:
(Assume the invariant is true at the beginning of the loop,
show the invariant is true at the end of the loop for the updated variables)
Assume : SumList(@V) + @C = SumList(L)
Let @V = H|T, @V' = T, @C' = H + @C
  SumList(@V') + @C'

= SumList(T) + (H + @C)	By substitution
= (SumList(T) + H) + @C	By associativity of addition
= SumList(H|T) + @C	By definition of SumList
= SumList(@V) + @C	By substitution
= SumList(L)	By assumption

Exiting Loop:
(Assume the invariant is true and the loop condition is false,
show that Out contains the desired value, L1 ++ L2)
Assume: SumList(@V) + @C = SumList(L), @V = nil
SumList(nil) + @C = @C = SumList(L)

I have shown that the invariant is true before and after the loop. **End proof…**