Madhuri Pyreddy
26 November 2019

CSCI 117 Lab 14

Part 1

Question 1:

a.
```
{Browse {Eval minus(7 10)}}
{Browse {Eval plus(plus(5 5) 10)}}
{Browse {Eval times(6 11)}}
```

When *** S *** is replaced, the procedure operations in the stack get popped out and no statements get executed. The reason is that there is no case function for minus in the Eval function. According to try-catch statement, when an operation runs into an error, the next operations following that will pop out of the stack. In this case, because there is no case function for minus, the procedure runs into an error. The next procedures following that ({Browse {Eval plus(plus 5 5) 10) } } and {Browse {Eval times (6 11) } }) are going to be popped off the stack.

b.
```
{Browse {Eval plus(plus(5 5) 10)}}
{Browse {Eval divide(6 0)}}
{Browse {Eval times(6 11)}}
```

When *** S *** is replaced, the procedure {Browse {Eval plus(plus 5 5) 10) } } is going to be passed into the function because there is a case function for plus, so it is getting passed. But, {Browse {Eval divide(6 0) } } is not getting passed because 6 is dividing by 0, and you cannot do that since it gives an error. So, it raises an error on {Browse {Eval divide(6 0) } }. Later on, {Browse {Eval times(6 11) } } doesn't get passed so it gets popped out of the stack in accordance with the try-catch statement.

Question 2:
It is not possible for the program to raise an error that is not caught by either of the two catch statements because when you get an error on the first catch statement, it doesn't get passed so it pops out the next 2 statements in accordance with the try-catch statement.

Part 2

Question 1: Call by variable is a special case of call by reference. The identity of the cell is passed to the procedure. For the function swap, you are passing in pointers for A and B. The cells A and B are stored into the parser. Call by value is a special case in which the value is passed to the procedure and put into a cell local to the procedure. The implementation is free either to copy the value or to pass a reference, as long as the function cannot change the value in the calling environment. For example, in the function swap, the variables' (A and B) values are not changed in the calling environment.

Call by Variable:
local Swap A B in

```
    proc {Swap A B}
 T in
       T = @A
     A := @B
      B := T
    end
local X Y in
     X = newCell 3
      Y = newCell 5
      {Swap X Y}
       A = @X
       B = @Y
      skip Browse A
      skip Browse B
     end
end
```

Call by Value:

```
local Swap A B in
    proc {Swap A B}
   X = newCell A
    Y = newCell B
    T Temp in
     T = @X
    Temp = @Y
    X := @Temp
     Y := @T
   end
   local X Y in
   X = newCell 23
   Y = newCell 25
    {Swap X Y}
      A = @X
      B = @Y
    skip Browse A
     skip Browse B
  end
end
```

Question 2: Call by value-result is a modification of call by variable. When the procedure is called, the content of a cell (i.e., a mutable variable) is put into another mutable variable local to the procedure Reverse. When the procedure Reverse returns, the content of the latter is put into the former.

Call By Value-Result:

```
local Reverse C D in
proc {Reverse A}
 X = newCell @A
  B = @X
  Rs = newCell nil
ReverseH in
proc {ReverseH L}
case L of nil then skip Basic
      [] '|'(1:H 2:T) then Rs := (H|@Rs) {ReverseH T}
      end
    end
    {ReverseH B}
    A := @Rs
  end
C = newCell [1 2 3 4]
{Reverse C}
D = @C
 skip Browse D
end
```

Question 3:

a. Call by name creates a procedure value for each argument. A procedure value used in this way is called a thunk. Each time the argument is needed, the procedure value is evaluated. It returns the name of a cell, i.e., the address of a mutable variable. Call by need is a modification of call by name in which the procedure value is evaluated only once. Its result is stored and used for subsequent evaluations.

   Call by Name:

   ```
   proc {Pow5 A}
       A := (@A * @A * @A *@A *@A)
   end
   local C = {newCell 0}
   C := 4
   {Pow5 fun {$} C end}
    {Browse @C} end
   ```

   Call by Need:

   ```
     proc {Pow5 A}
      B = {A} in
      B = @B * @B * @B * @B * @B
   end
   ```

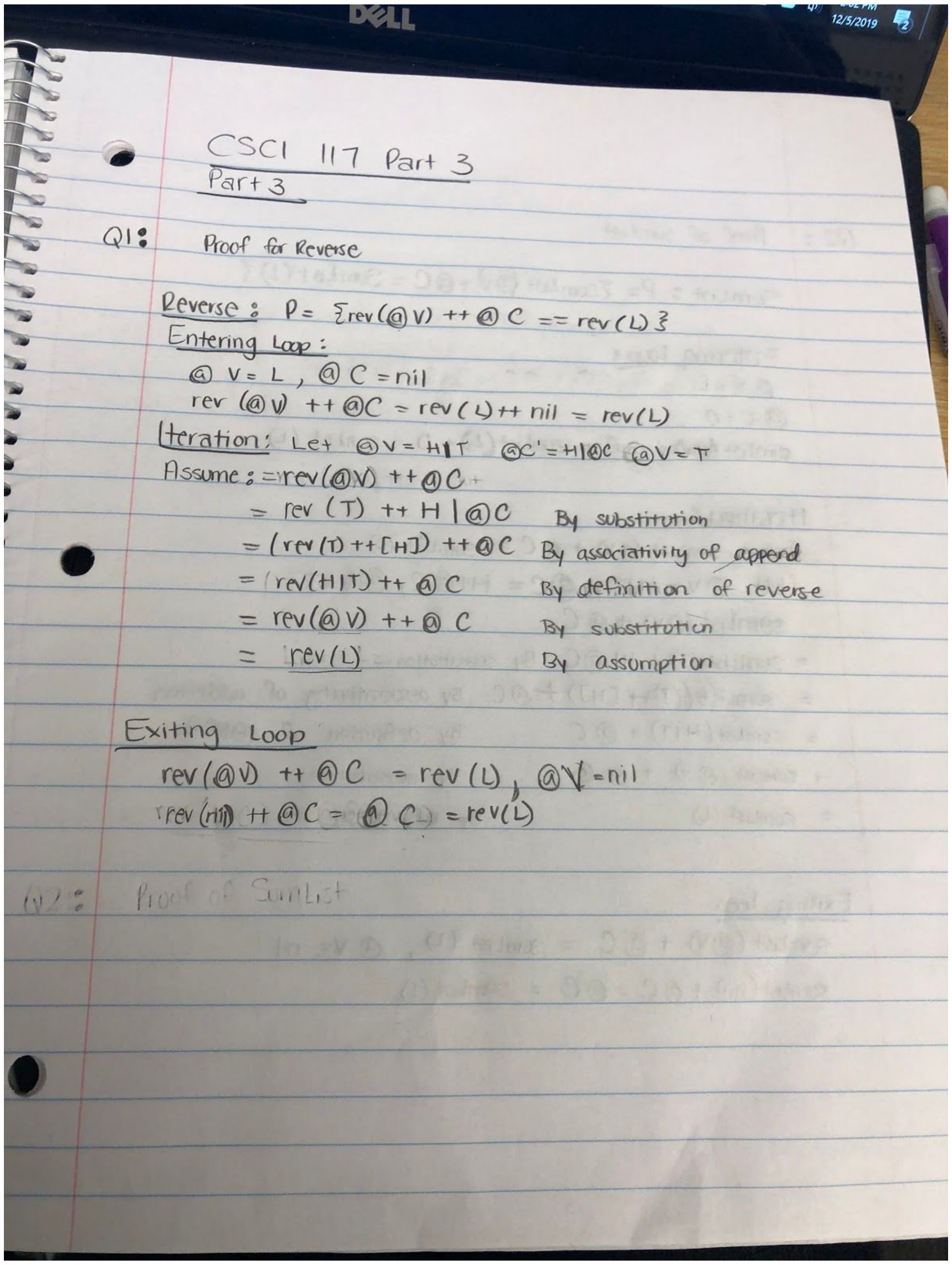
```
local C = {newCell 0}
C := 2
{Pow5 fun {$} C end}
 {Browse @C} end
```

b. Call by need is more efficient.
c. Call by name is slower than call by need by 5-10% because it creates a thunk for each argument whereas call by need allows the “thunk” to be evaluated only once in the Fib function. In other words, call by value allows the Fibonacci value to evaluate every time the value’s needed within the function. {Fib 11} will be 10 times faster in call by name than in call by need.
d. Call by Name:

```
proc {SubtractTwenty A}
while (@A > 20)
{A}:=@{A} - 20
end
local C={NewCell 0} in
C:=25
{SubtractTwenty fun {$} C end}
{Browse @C}
end
```

In call by name, we keep changing value of A to check the while condition. With Call By Name, we get the updated value every time when subtracting 20. However, with call by need, we need extra assignment statements when subtracting 20 with the updated value.

Part 3



CSCI 117 Part 3

Part 3

Q1: Proof for Reverse

Reverse: P = {rev(@V) ++ @C == rev(L)}

Entering Loop:

@V = L , @C = nil

rev(@V) ++ @C = rev(L) ++ nil = rev(L)

Iteration: Let @V = H|T @C' = H|@C @V = T

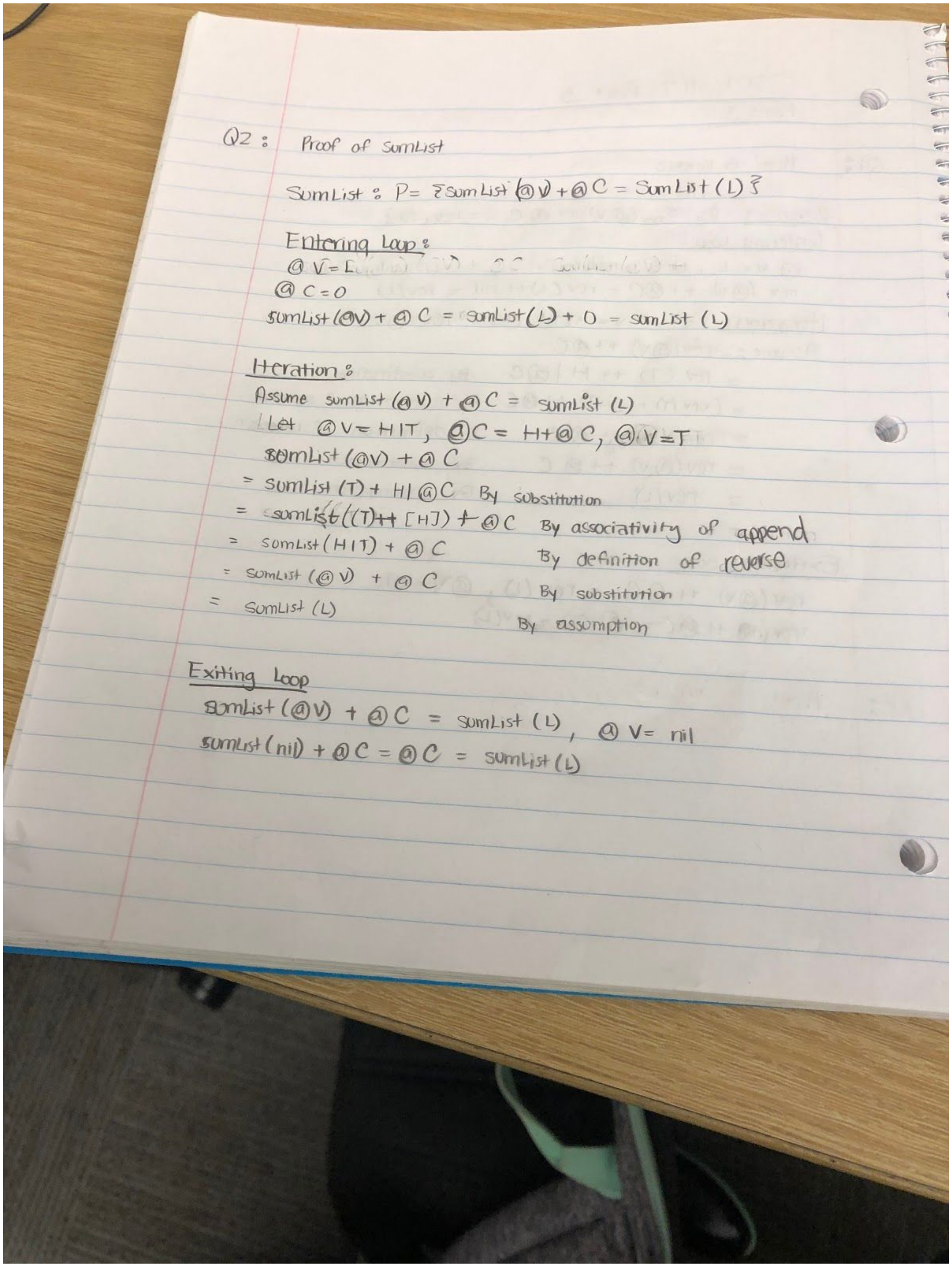
Assume: = rev(@V) ++ @C

= 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

rev(@V) ++ @C = rev(L), @V = nil

rev(nil) ++ @C = @C = rev(L)



Q2 : Proof of sumList

SumList : P= { SumList (@V) + @C = SumList (L) }

<u>Entering Loop</u> :

@ V = L

@ C = 0

sumList (@V) + @ C = sumList (L) + 0 = sumList (L)

<u>Iteration</u> :

Assume sumList (@ V) + @ C = sumList (L)

Let @ V = H | T , @ C = H + @ C , @ V = T

sumList (@V) + @ C

= sumList (T) + H | @ C By substitution

= sumList ((T) ++ [H]) + @ C By associativity of append

= sumList (H | T) + @ C By definition of reverse

= sumList (@ V) + @ C By substitution

= sumList (L) By assumption

<u>Exiting Loop</u>

sumList (@V) + @ C = sumList (L) , @ V = nil

sumList (nil) + @ C = @ C = sumList (L)