Introduction to Programming Exercises

Annalise Tarhan

April 11, 2020

1 A Calculator

1.1 Calculate the value of 2^{100} without using any new functions and without typing a hundred 2s.

Using the notation T5X to mean two five times, or 2^5,

```
declare
T5X=2*2*2*2*2
declare
T25X=T5X*T5X*T5X*T5X*T5X
declare
T100X=T25X*T25X*T25X*T25X

{Browse T100X}
```

displays the correct answer, 1267650600228229401496703205376.

1.2 Calculate the exact value of 100! without using any new functions. Are there any possible shortcuts in this case?

Assuming 'new functions' doesn't include functions given in the text,

```
declare
fun {Fact N}
    if N==0 then 1 else N*{Fact N-1} end
end

{Browse {Fact 100}}
```

displays the correct answer, 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000. Otherwise, each number between 1 and 100 would have to be typed manually. Unlike the previous example, all of the factors are unique, so the calculation cannot be simplified. (Short of breaking the factors down into their factors, which is hardly simpler.)

2 Calculating Combinations

2.1 Write a more efficient combinations function using the given alternative combinations definition.

```
declare
fun {FallFact N K}
   if N-K<1   % Handles the case where N==0
   then 1
   else N*{NewFact N-1 K}
   end
end

declare
fun {NewComb N K}
   {FallFact N N-K} div {FallFact K 1}
end
```

2.2 Use the fact that $\binom{n}{k} = \binom{n}{n-k}$ to write an even more efficient function.

```
declare
fun {NewComb N K}
   if K>N div 2
   then {FallFact N K} div {FallFact N-K 1}
   else {FallFact N N-K} div {FallFact K 1}
   end
end
```

3 Program Correctness

3.1 Show that the Pascal function is correct.

```
declare Pascal AddList ShiftLeft ShiftRight fun {Pascal N}
    if N==1 then [1] else
        {AddList
            {ShiftLeft {Pascal N-1}}
            {ShiftRight {Pascal N-1}}}
    end
end
```

Base case: N==1 returns [1].

Inductive step: For an arbitrary N, assume that Pascal returns the correct list of size N+1. Then, for N+1, Pascal will return a list of size N+2. The first and last items will be 1 because each will be the sum of a zero added by the ShiftLeft/ShiftRight functions and the 1s at the beginning and end of the (correct) result returned by Pascal N. Each of the N entries between them will be the sum of the Nth and N+1th digit of Pascal N, which satisfies the definition of a Pascal triangle.

4 Program Complexity

4.1 What does section 1.7 say about programs whose time complexity is a high-order polynomial? Are they practical or not?

Section 1.7 says that time complexity proportional to a polynomial function in n is preferable to an exponential function in n and that programs whose time complexity is a low-order polynomials are practical. Programs whose time complexity is a high-order polynomial are still better than those with exponential complexity, but impractical except for very small inputs.

5 Lazy Evaluation

5.1 Ints is defined as a function that lazily calculates an infinite list of integers. For the following function, what happens if we call SumList Ints 0? Is this a good idea?

```
fun {SumList L}
case L of X|L1 then X+{SumList L1} else 0 end
end
```

Even though Ints calculates lazily, SumList will keep asking for the "tail" of {Ints 0} indefinitely, so it will run until it causes a stack overflow. Probably not a good idea.

6 Higher-Order Programming

6.1 Calculate individual rows of Pascal's triangle using subtraction, multiplication, and other operations.

Figure 1: Subtraction

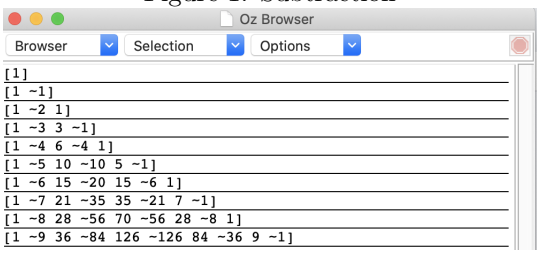


Figure 2: Multiplication

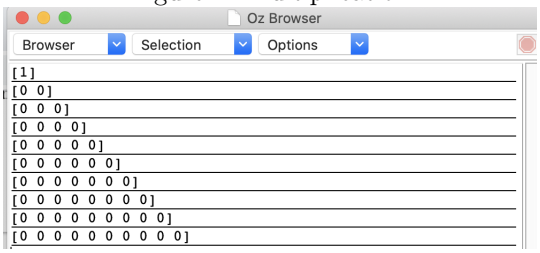
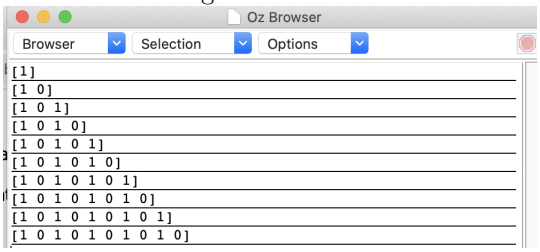


Figure 3: Power



6.2 Why does using multiplication give a triangle with all zeros? Try the following kind of multiplication instead: fun {Mul1 X Y } (X+1) * (Y+1) end. What does the 10th row look like when calculated with Mul1?

Except for the single 1 in the first row, each element on every row has an "ancestor" that was multiplied by one of the zeros used to pad rows in the ShiftLeft and ShiftRight operations.

Using Mul1, the largest numbers on the 10th row have blown up to over 70 digits.

7 Explicit State

7.1 Explain what Browse displays for each of the following fragments. In the first, X refers to two different variables. In the second, X refers to a cell.

```
local X in
      X=23
      local X in
            X=44
      end
      {Browse X}
end
```

Here, X refers to the variable referencing 23, since that is the only variable named X within the scope of Browse. The scope of the second X began with the *in* following its declaration and ended with the *end* before Browse.

```
local X in
      X={NewCell 23}
      X:=44
      {Browse @X}
end
```

In this case, Browse will display 44. The cell was created with the value 23, but in the next line was assigned the value of 44.

8 Explicit State and Functions

8.1 What is wrong with the following definition of an Accumulator? It should behave intuitively, adding together all the arguments of all calls. How should it be written?

```
declare
fun {Accumulate N}
Acc in
      Acc={NewCell 0}
      Acc:=@Acc+N
      @Acc
end
```

The problem is that Accumulate creates a new cell every time it is called. It gives it an initial value of zero and adds the input, but has no way of tracking the inputs from previous or future calls.

```
declare
Acc={NewCell 0}
fun {Accumulate N}
      Acc:=@Acc+N
      @Acc
end
```

In this corrected definition, the cell is instantiated outside the function

9 Memory Store

9.1 Use the given memory store to write an improved version of FastPascal that remembers previously calculated rows.

```
declare
Store={NewStore}
{Put Store 1 [1]}
fun {FasterPascal N}
   if N>{Size Store} then
      for I in {Size Store}+1..N do
         {Put Store I {AddList
            {ShiftLeft {Get Store I-1}}
            {ShiftRight {Get Store I-1}}}}
      end
   end
   {Get Store N}
end
```

9.2 Rewrite the memory store using memory cells.

```
declare
fun {CellStore}
   C={NewCell nil}
   Size={NewCell 0}
   proc {Put K V}
      if {GetKInC K @C} == 'not present' then
         C:=K|V|@C
         Size:=@Size+1
      end
   end
   fun {Get K}
      {GetKInC K @C}
   end
   fun {GetKInC K List}
      case List of Key|Value|Rest then
         if K==Key then Value else {GetKInC K Rest}
         end
      else
         'not present'
      end
   end
   fun {GetSize}
```

```
            @Size
      end
in
      cellstoreobject(put:Put get:Get size:GetSize)
end

proc {Put CS K V} {CS.put K V} end

declare
fun {Get CS K} {CS.get K} end

declare
fun {Size CS} {CS.size} end
```

10 Explicit State and Concurrency

10.1 The given code uses a cell to store a counter and increments in in two threads. Execute it several times. Do you ever get the result 1? Why could this be?

I executed the code several thousand times in a loop, and only ever got 2.

10.2 Modify the example by adding calls to Delay in each thread to create a scheme that always results in 1.

```
declare
fun {TestConcurrency}
      local C in
            C={NewCell 0}
            thread I in
                  I=@C
                  {Delay 5}
                  C:=I+1
            end
            thread J in
                  J=@C
                  {Delay 5}
                  C:=J+1
            end
            {Delay 10}
            @C
      end
end
```

```
declare
OneCount={NewCell 0}

declare
TwoCount={NewCell 0}

for I in 1..10 do
     local Result in
          Result = {TestConcurrency}
          if Result==1 then OneCount:=@OneCount+1 end
          if Result==2 then TwoCount:=@TwoCount+1 end
     end
end

{Browse 'Ones:'}
{Browse @OneCount}
{Browse 'Twos:'}
{Browse @TwoCount}
```

10.3 Consider the version of the counter that never gives the result 1. What happens if you use the delay technique to try to get a 1 anyway?

Because there is a locking mechanism, the threads are prevented from starting until the other thread has finished, so it always returns 2 anyway. The delays just make it take longer.