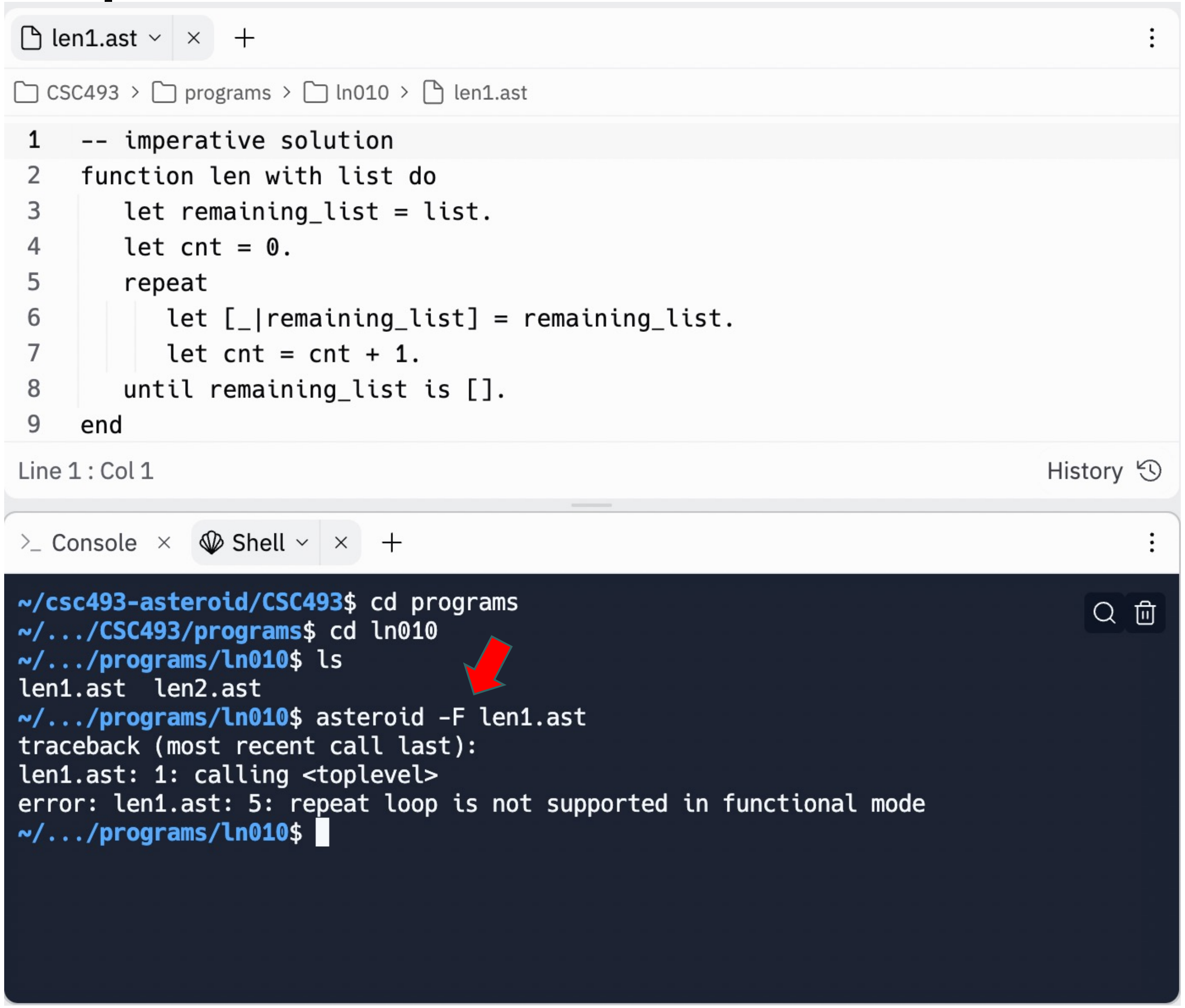
Functional Programming in Asteroid

- Asteroid supports a functional sublanguage largely inspired by ML.
- You can turn the Asteroid interpreter into a functional language interpreter with the ‘-F’ flag
  - In this mode imperative statements will be rejected with some exceptions
  - Most notable exception is the let statement – we’ll discuss this later

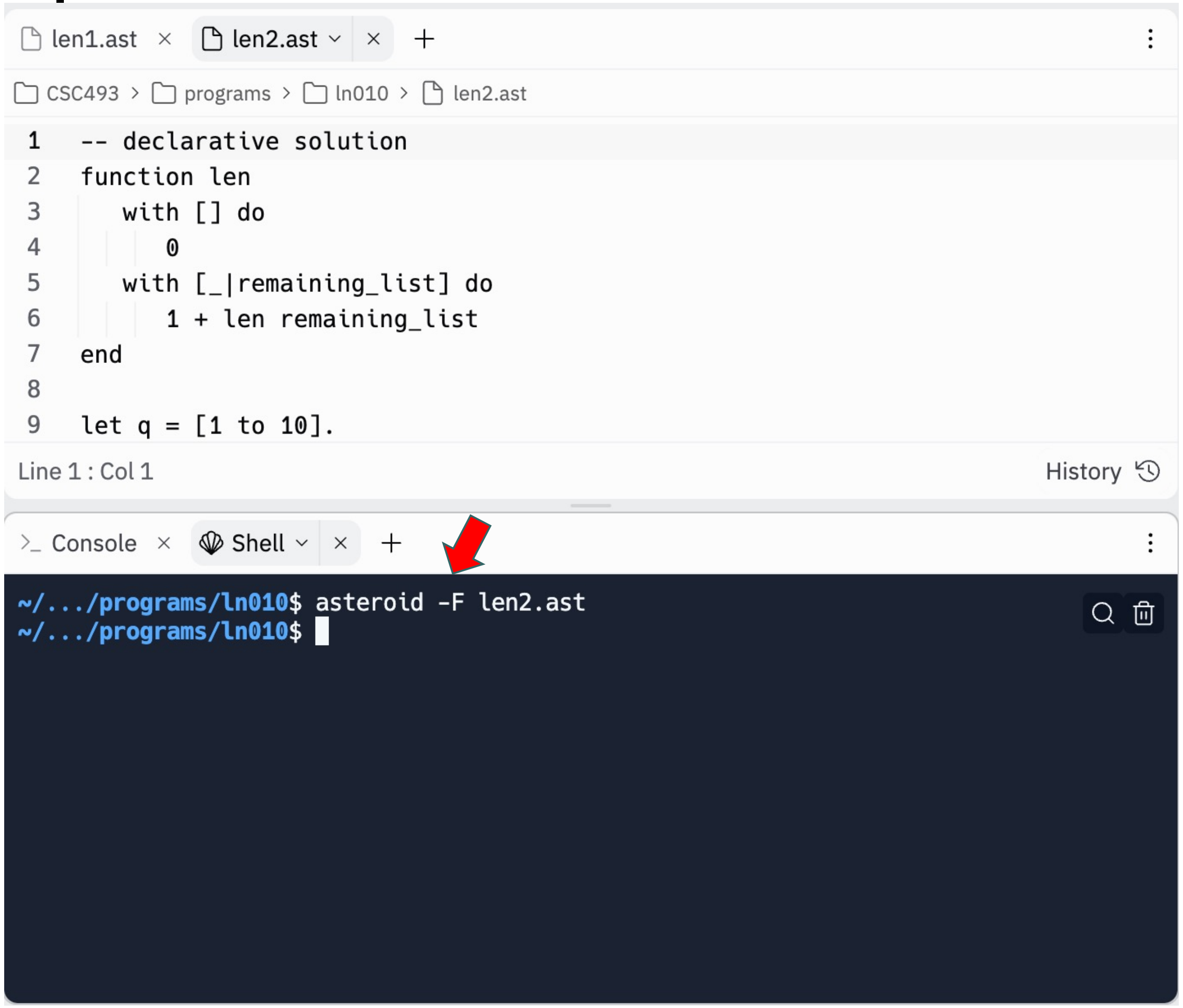
Functional Programming in Asteroid



```
-- imperative solution
function len with list do
    let remaining_list = list.
    let cnt = 0.
    repeat
        let [_|remaining_list] = remaining_list.
        let cnt = cnt + 1.
    until remaining_list is [].
end
```

```
~/csc493-asteroid/CSC493$ cd programs
~/.../CSC493/programs$ cd ln010
~/.../programs/ln010$ ls
len1.ast  len2.ast
~/.../programs/ln010$ asteroid -F len1.ast
traceback (most recent call last):
len1.ast: 1: calling <toplevel>
error: len1.ast: 5: repeat loop is not supported in functional mode
~/.../programs/ln010$
```

Functional Programming in Asteroid



```
-- declarative solution
function len
   with [] do
      0
   with [_|remaining_list] do
      1 + len remaining_list
end

let q = [1 to 10].
```

```
~/.../programs/ln010$ asteroid -F len2.ast
~/.../programs/ln010$ 
```

Lambda Functions

- The most recognizable feature of the functional programming paradigm is the lambda function
  - Virtually every programming language designed in the last decade or two supports lambda functions – by extension, they support the functional programming paradigm (even if limited)

Swift

```
func main() {
    let y: (Int) -> Int = { x in x + 1 }
}
```

Rust

```
fn main() {
    let x: fn(i32) -> i32 = |x| x + 1;
}
```

Python

```
def main():
    y = lambda x: x + 1
```

Go

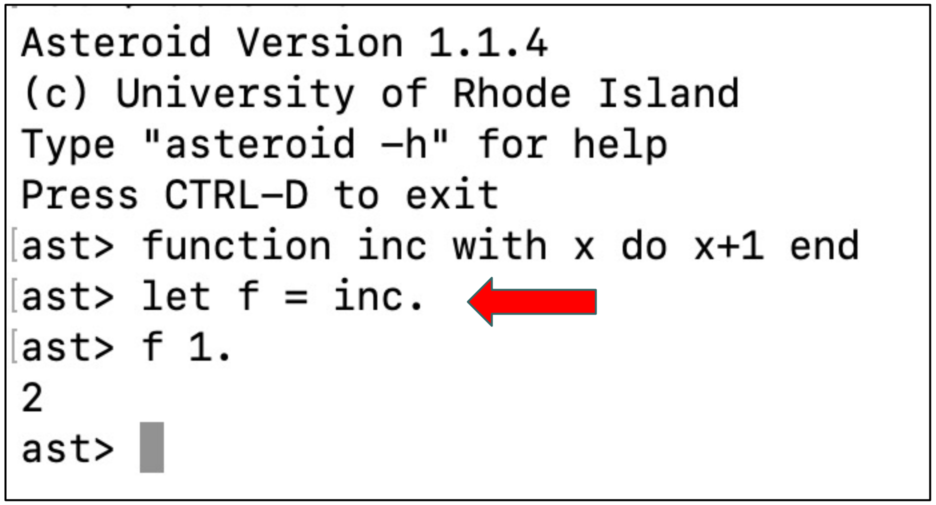
```
func main() {
    y := func(x int) int {
        return x + 1
    }
}
```

Asteroid

```
function main with () do
    let y = lambda with x do x+1.
end
```

Lambda Functions

- The implication of the support of lambda functions is that functions are considered first-class citizens,
  - ☞ They are Values!
- Consider,



```
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
ast> function inc with x do x+1 end
ast> let f = inc.
ast> f 1.
2
ast>
```

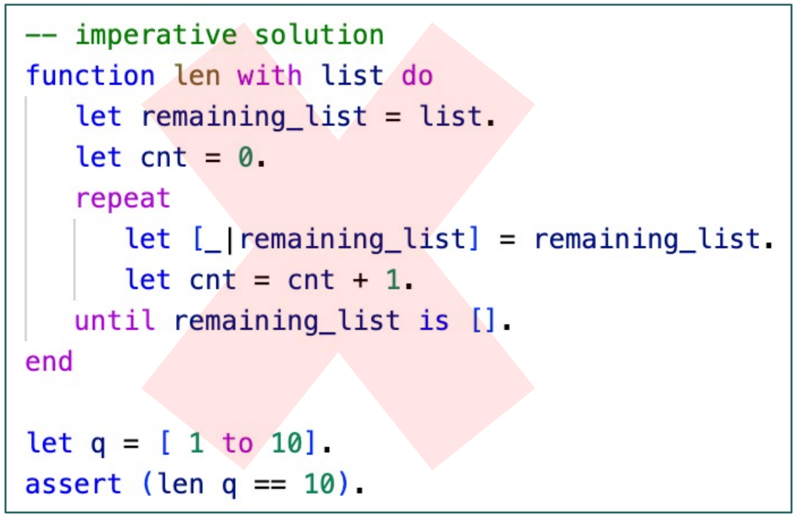
We can copy function values like any other value!

Other Characteristics of Functional Programming

- No iteration – only recursion.
- No if statements – only if expressions.
- ”Single valued variables”
  - Variables are shorthand notations for expression values

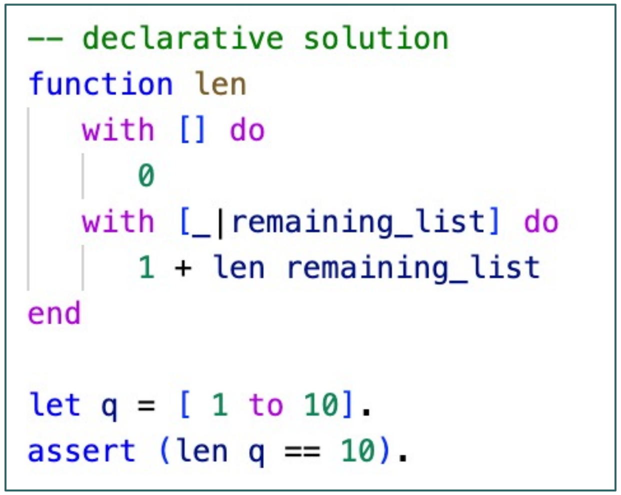
No Iteration

- Iteration is not supported
- Data structures must be traversed with recursion
  - Recursive functions with multi-dispatch!



```
-- imperative solution
function len with list do
   let remaining_list = list.
   let cnt = 0.
   repeat
      let [_|remaining_list] = remaining_list.
      let cnt = cnt + 1.
   until remaining_list is [].
end

let q = [ 1 to 10].
assert (len q == 10).
```

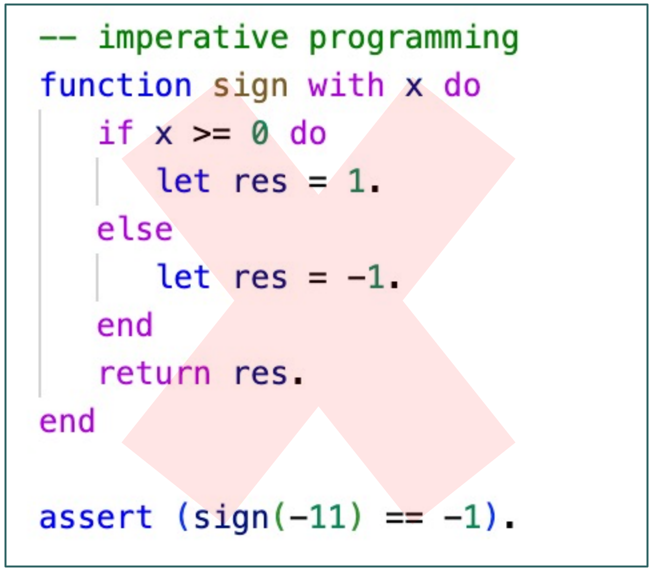
vs



```
-- declarative solution
function len
   with [] do
      0
   with [_|remaining_list] do
      1 + len remaining_list
end

let q = [ 1 to 10].
assert (len q == 10).
```

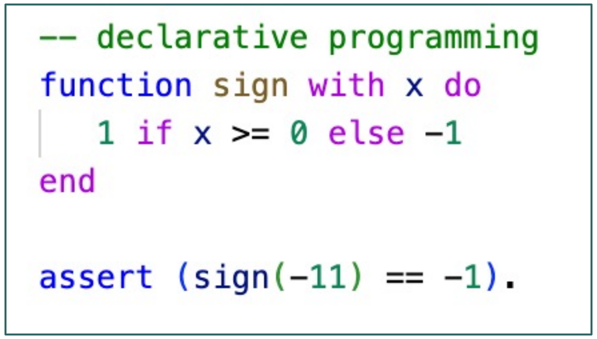
No If Statements

- If statements are designed to inherently modify machine state and therefore are not allowed in functional programming
- We use if expressions instead
  - Also fits better into the notion of “everything is a value”



```
-- imperative programming
function sign with x do
    if x >= 0 do
        let res = 1.
    else
        let res = -1.
    end
    return res.
end

assert (sign(-11) == -1).
```

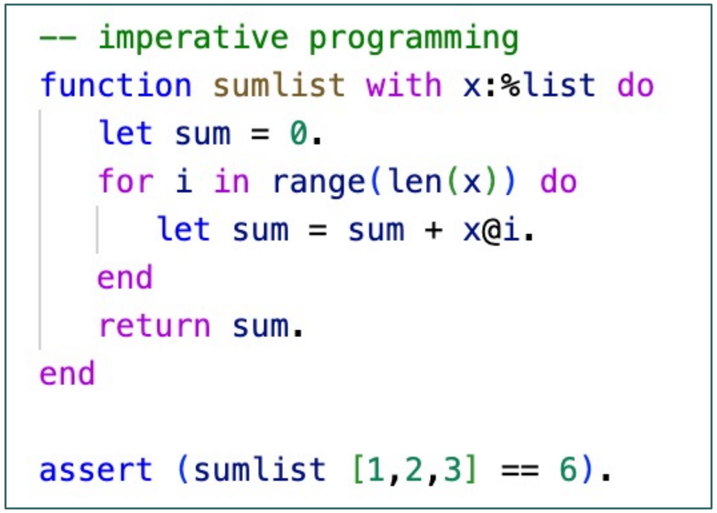
vs



```
-- declarative programming
function sign with x do
    1 if x >= 0 else -1
end

assert (sign(-11) == -1).
```

Single Valued Variables

- In imperative programming variables maintain the machine state,



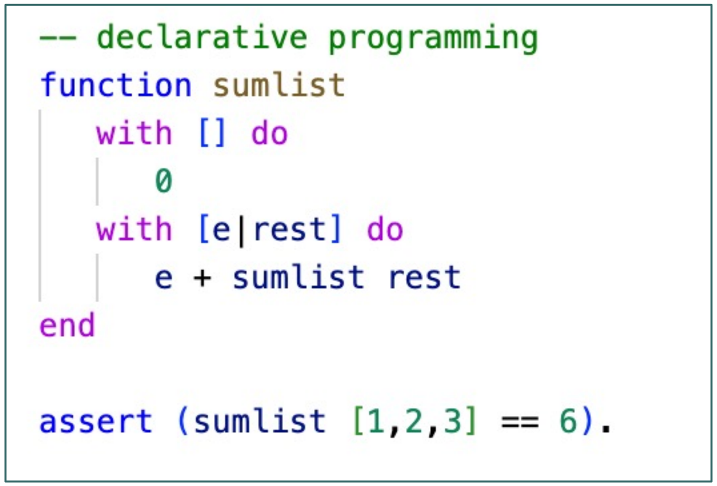
```
-- imperative programming
function sumlist with x:%list do
   let sum = 0.
   for i in range(len(x)) do
      let sum = sum + x@i.
   end
   return sum.
end

assert (sumlist [1,2,3] == 6).
```

The variable sum is updated iteratively and at each iteration contains the partial solution computed so far.

Note that the evolution of the values stored in sum depends on the length of the input list!

Single Valued Variables

- In functional programming variables act like a shorthand notation for a single value (per function call)

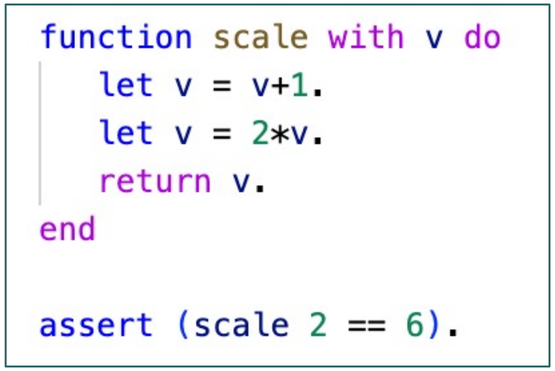


```
-- declarative programming
function sumlist
    with [] do
        0
    with [e|rest] do
        e + sumlist rest
end

assert (sumlist [1,2,3] == 6).
```

Here e and rest contain a single value (per function call) that does not change throughout that function call.

Single Valued Variables

- Even if we assign multiple values to the same variable, it still has the flavor of a value shorthand notation
  - We don’t have iteration to evolve the value further than the given assignments



```
function scale with v do
    let v = v+1.
    let v = 2*v.
    return v.
end

assert (scale 2 == 6).
```

Here we use the multiple assignments to v to break the expression computation,

```
return 2*(v+1)
```

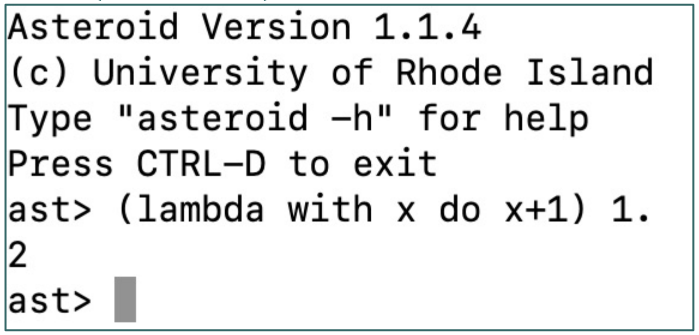
Into simpler computational steps.

Functional Programming in Asteroid

- Let’s see how the programs that we developed in the lambda calculus translate into Asteroid
  - Should be straight forward since Asteroid supports the functional programming paradigm.

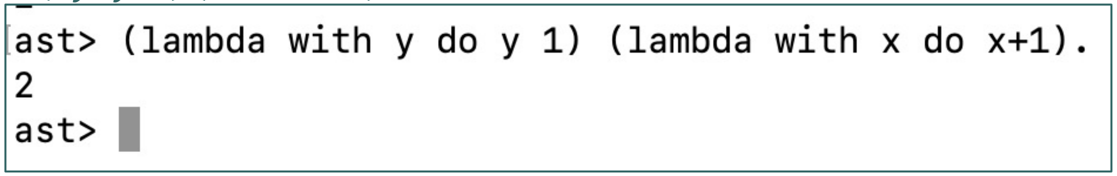
Original Lambda Examples

$(\lambda x.\, x + 1)\; 1 \;\Rightarrow 2$



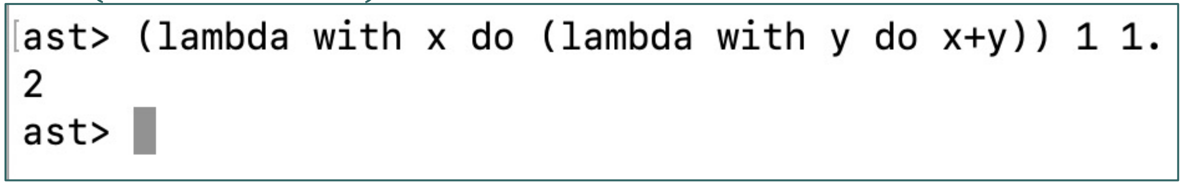
```
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
ast> (lambda with x do x+1) 1.
2
ast> 
```

$(\lambda y.\, y\; 1)(\lambda x.\, x + 1) \Rightarrow 2$



```
ast> (lambda with y do y 1) (lambda with x do x+1).
2
ast> 
```

$(\lambda x. (\lambda y.\, x + y))\; 1\; 1 \Rightarrow 2$



```
ast> (lambda with x do (lambda with y do x+y)) 1 1.
2
ast> 
```

Classic Functional Programming

- Let’s look at some classic functional programming examples
- The most noticeable issue of course is that data structures like lists are accessed in a sequential manner with the head-tail pattern using recursion.

Sum/Mult

- Sum/multiply all the elements of a list.

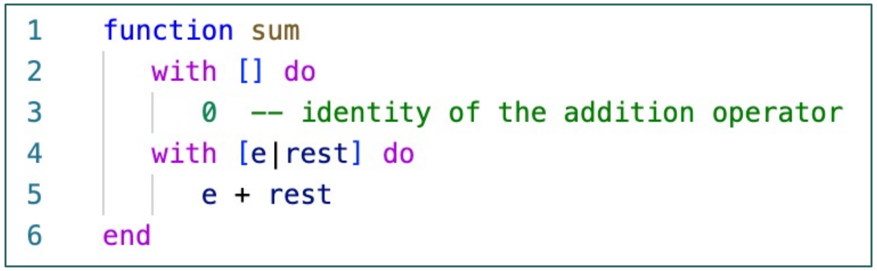
```
function sum
   with [] do
      0  -- identity of the addition operator
   with [e|rest] do
      e + rest
end
```

```
function mult
   with [] do
      1  -- identity of the multiplication operator
   with [e|rest] do
      e * rest
end
```

Sum/Mult

- Identity means it is a value that if added/multiplied to another value returns the original value, e.g
  - 2+0 = 2
  - 2*1 = 2
- In functional algorithm design identity values are often important as part of the **recursion base cases**.

Sum/Mult



```
function sum
  with [] do
    0  -- identity of the addition operator
  with [e|rest] do
    e + rest
end
```

- Consider sum [1,2,3]
  - 1 + sum [2,3]
  - 1 + 2 + sum [3]
  - 1 + 2 + 3 + sum []
  - 1 + 2 + 3 + 0

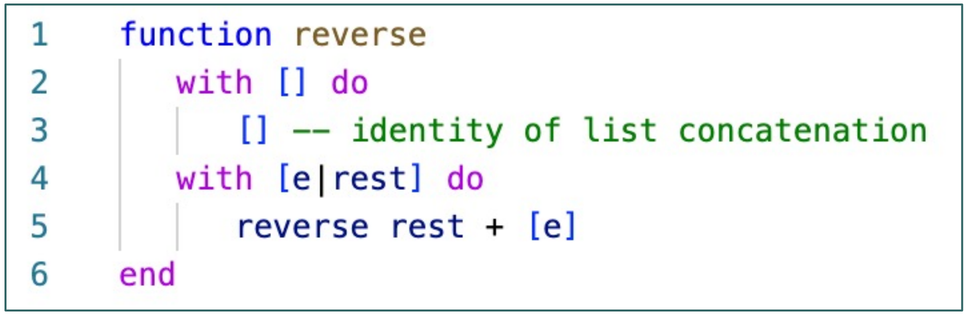
String Concatenation

- If we consider the + operator to work as a string concatenation operator,
  - “abc” + “edf” = “abcdef”
- What is the identity of string concatenation?

Reverse

- Given a list of values the reverse function reverses that list, e.g.
  - reverse [1,2,3] = [3,2,1]
- Assume that the + operator functions as a list concatenation operator, what is the identity of + as a list concatenation operator?

Reverse



```
function reverse
  with [] do
    [] -- identity of list concatenation
  with [e|rest] do
    reverse rest + [e]
end
```

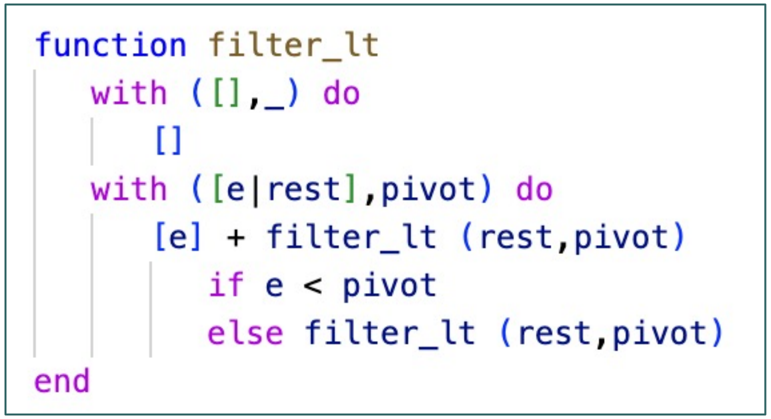
- Note the empty list as the identity of list concatenation.

Filter

- Another classic functional programming algorithm is ‘filter’,
  - *Given a list of values, return a list of values that are smaller/larger than a given pivot value.*
- For example,
  - filter_lt ([1,2,3,4,5],4) = [1,2,3]

Filter

- Again we can observe that the base case is the identity of the fundamental operation in the recursive case: list concatenation
- Also, in keeping with with declarative programming we see that we “declare” what we want to do with each input configuration.



```
function filter_lt
   with ([],_) do
      []
   with ([e|rest],pivot) do
      [e] + filter_lt (rest,pivot)
         if e < pivot
         else filter_lt (rest,pivot)
end
```

QuickSort

- Sort a list according to the quicksort algorithm - recursive partitioning according to a pivot value,
  - qsort [3,1,2] = [1,2,3]
- In the declarative setting this algorithm is straight forward.

QuickSort

```
function qsort
  with [] do
    []
  with [pivot|rest] do
    let less = filter_lt (rest,pivot).
    let more = filter_ge (rest,pivot).
    qsort less + [pivot] + qsort more.
end
```

```
function filter_lt
   with ([],_) do
      []
   with ([e|rest],pivot) do
      [e] + filter_lt (rest,pivot)
         if e < pivot
         else filter_lt (rest,pivot)
end
```

```
function filter_ge
   with ([],_) do
      []
   with ([e|rest],pivot) do
      [e] + filter_ge (rest,pivot)
         if e >= pivot
         else filter_ge (rest,pivot)
end
```