Currying – Computing with Partially Evaluated Functions

- The idea of higher-order programming and lambda functions gives rise to the idea of partially evaluated functions.
- Again, we can look at the lambda calculus for foundations

Currying

- Consider a lambda expression that takes a pair of values and adds them together.
- Now assume that both arguments are not immediately available…only one at a time is available
  - I know, it's a stretch but bear with me…
- We can rewrite the lambda expression to deal with that situation by **computing partially evaluated lambda expressions**.

Currying

- Here is the original lambda expression expecting a pair of values,

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

← Single Value

- Here is a lambda expression that takes one value at a time,

$$\left(\lambda x. (\lambda y. x + y)\right) 1\ 2$$

← Multiple Values

- Note that after taking in the first argument it computes a partially evaluated function that expects the second argument.

Currying

- Let’s take a look how the computation of the two lambda expressions differ,

$$(\lambda(x,y).x+y)(1,2) \Longrightarrow x+y[(x,y) \longleftarrow (1,2)]$$
$$\Longrightarrow x+y[x \leftarrow 1, y \leftarrow 2] \Longrightarrow 1+2 \Longrightarrow 3$$

$$\big(\lambda x.(\lambda y.x+y)\big)1\ 2 \Longrightarrow (\lambda y.x+y)[x \leftarrow 1]2$$
$$\Longrightarrow (\lambda y.1+y)2 \Longrightarrow 1+\mathrm{y}[y \leftarrow 2] \Longrightarrow 1+2 \Longrightarrow 3$$

Partially evaluated function

Currying

- This technique also applies to functions that take more than two values,

$$(\lambda(x, y, z).\, x + y + z)(1,2,3)$$

⬇

$$\left(\lambda x.\, \left(\lambda y.\, (\lambda z.\, x + y + z)\right)\right)\, 1\ 2\ 3$$

Currying

Dr Haskell Curry, mathematician and logician, 1900-1982

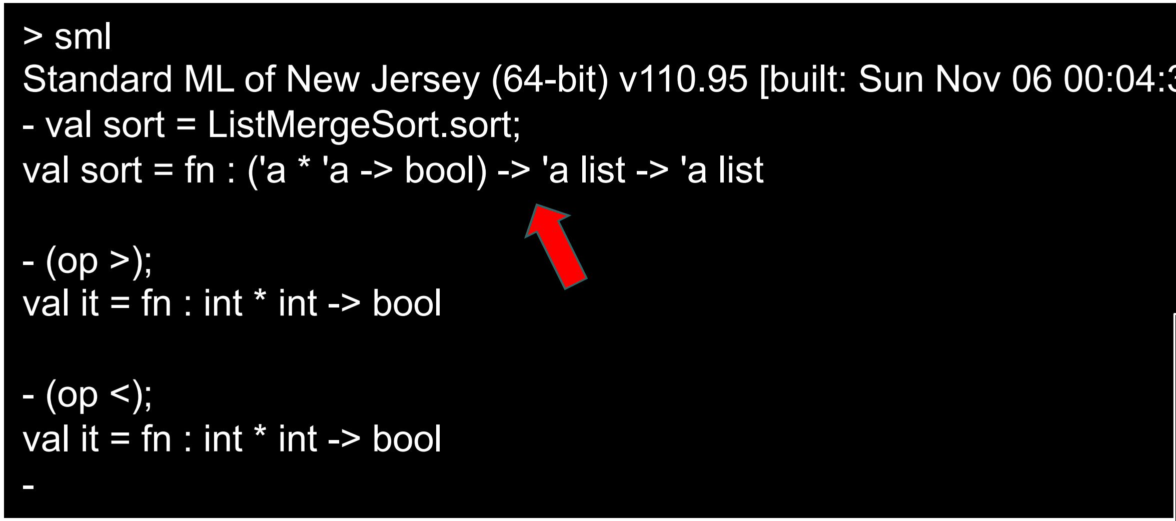
- This technique of turning a function expecting a tuple of values to a cascade of lambda functions is called **currying**.
- It was invented by the mathematician and logician Haskell Curry.
- He developed this technique while working on combinatory logic.

Functional Programming

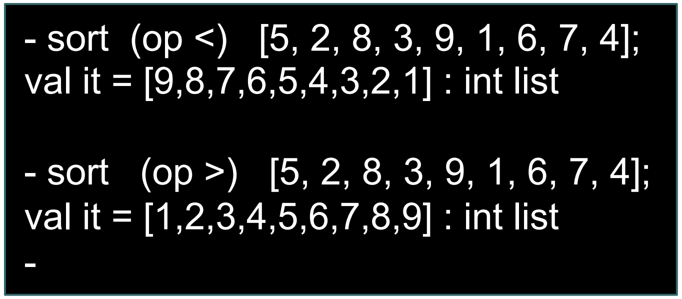
- Curried functions are important in the functional programming field because they make libraries for functional languages much more flexible.
- We can use partially evaluated library functions to define our own functions

SML

- Here is an example in SML taking advantage of the curried sort function.

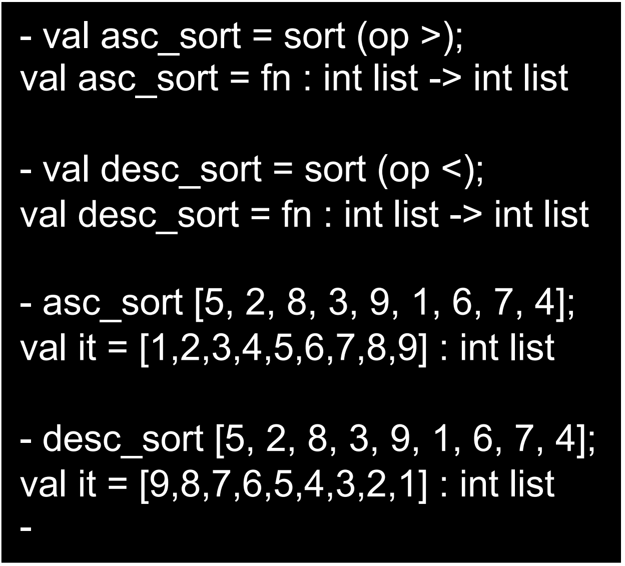


```
> sml
Standard ML of New Jersey (64-bit) v110.95 [built: Sun Nov 06 00:04:31 2022]
- val sort = ListMergeSort.sort;
val sort = fn : ('a * 'a -> bool) -> 'a list -> 'a list

- (op >);
val it = fn : int * int -> bool

- (op <);
val it = fn : int * int -> bool
-
```



```
- sort  (op <)   [5, 2, 8, 3, 9, 1, 6, 7, 4];
val it = [9,8,7,6,5,4,3,2,1] : int list

- sort   (op >)   [5, 2, 8, 3, 9, 1, 6, 7, 4];
val it = [1,2,3,4,5,6,7,8,9] : int list
-
```

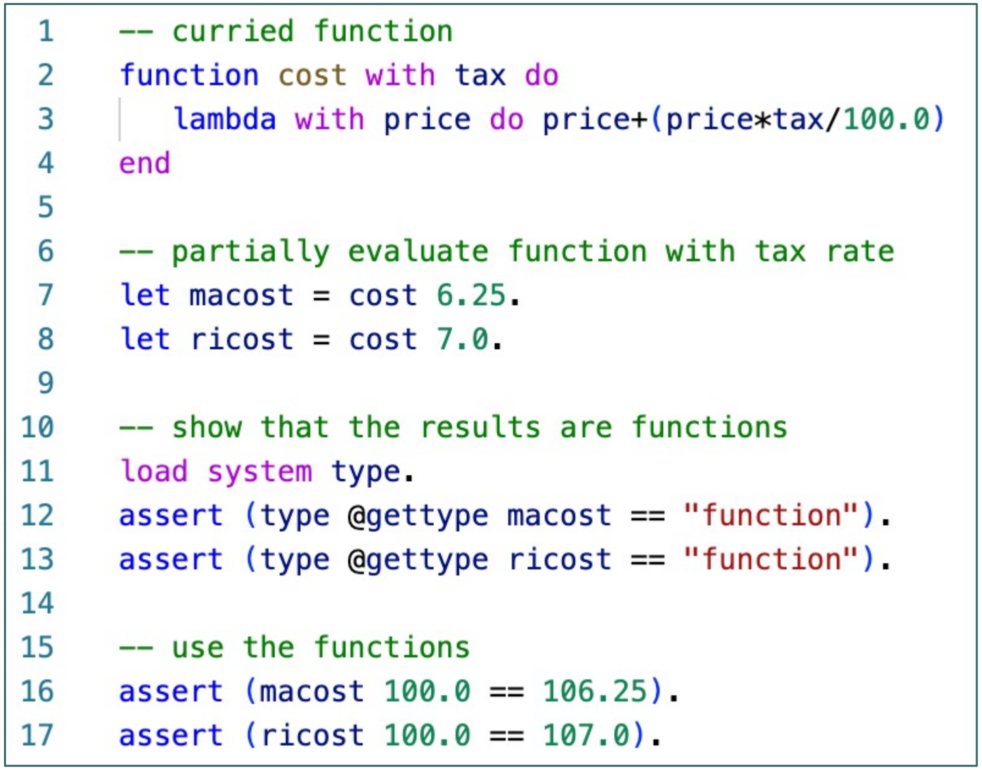
Partially evaluated functions



```
- val asc_sort = sort (op >);
val asc_sort = fn : int list -> int list

- val desc_sort = sort (op <);
val desc_sort = fn : int list -> int list

- asc_sort [5, 2, 8, 3, 9, 1, 6, 7, 4];
val it = [1,2,3,4,5,6,7,8,9] : int list

- desc_sort [5, 2, 8, 3, 9, 1, 6, 7, 4];
val it = [9,8,7,6,5,4,3,2,1] : int list
-
```

Asteroid

- Even though the modules and APIs are written in a more traditional, non-curried style in most modern programming languages, currying is still a powerful programming tool
- Here is a simple example written in Asteroid,

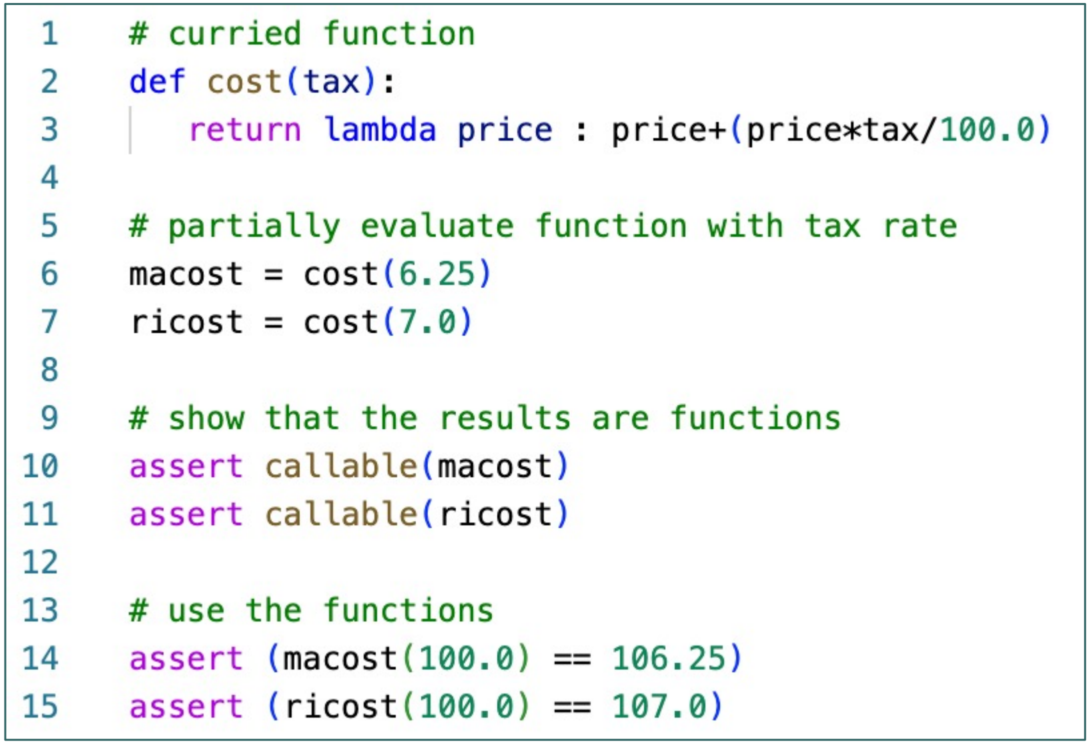


```
-- curried function
function cost with tax do
   lambda with price do price+(price*tax/100.0)
end

-- partially evaluate function with tax rate
let macost = cost 6.25.
let ricost = cost 7.0.

-- show that the results are functions
load system type.
assert (type @gettype macost == "function").
assert (type @gettype ricost == "function").

-- use the functions
assert (macost 100.0 == 106.25).
assert (ricost 100.0 == 107.0).
```

ln013/price.ast

Python

- Here is the same program written in Python

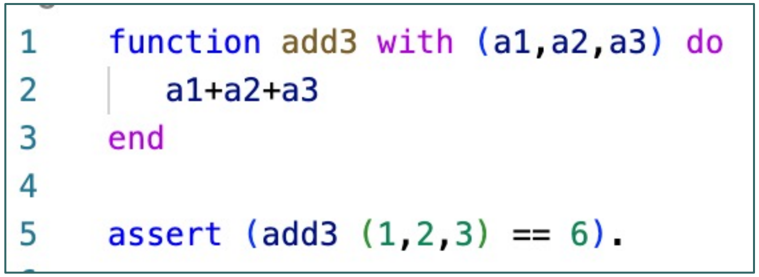


```python
# curried function
def cost(tax):
    return lambda price : price+(price*tax/100.0)

# partially evaluate function with tax rate
macost = cost(6.25)
ricost = cost(7.0)

# show that the results are functions
assert callable(macost)
assert callable(ricost)

# use the functions
assert (macost(100.0) == 106.25)
assert (ricost(100.0) == 107.0)
```

ln013/price.py

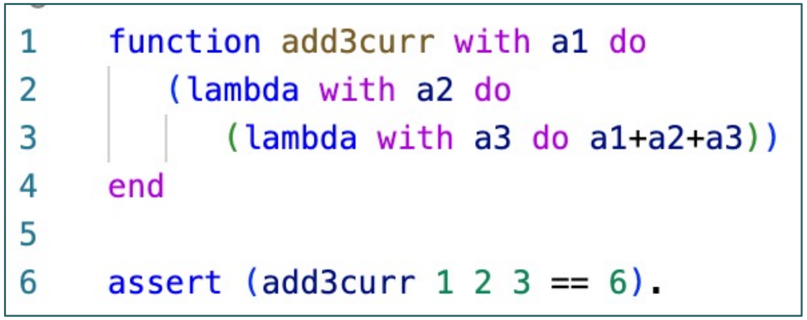
Currying of more than Two Arguments

- The return value is a cascade of lambda functions



```
function add3 with (a1,a2,a3) do
    a1+a2+a3
end

assert (add3 (1,2,3) == 6).
```

ln013/add3.ast



```
function add3curr with a1 do
    (lambda with a2 do
        (lambda with a3 do a1+a2+a3))
end

assert (add3curr 1 2 3 == 6).
```

ln013/add3curr.ast

Currying Function in other Languages

- Any language that supports lambda functions and static scoping supports function currying
- This includes pretty much all languages designed over the last decade or two,
  - Python, Rust, Swift, Go, Asteroid,…