Functions as First-Class Citizens

- Functions as first-class citizens means that we can pass functions around in a program as values, not much different than an integer or real value!
- When functional languages first appeared in the late 1970's and the 1980's this was a radical concept
- Today almost all modern languages support this, e.g.
  - Python, JavaScript, Rust, Go

Functions as First-Class Citizens

```
-- first-class functions

-- define our increment function
function inc with i do
    return i+1.
end

let foo = inc.  -- foo now holds a function value
let x = foo 1.  -- execute the function in foo with value 1
assert(x == 2).
```

Functions as First-Class Citizens

```
-- first-class functions

-- define our increment function
function inc with i do
    return i+1.
end

-- define our decrement function
function dec with i do
    return i-1.
end

-- c expects a function and a value
function c with (f:%function, v:%integer) do
    return f v.
end

-- we can modify c's behavior depending what kind of
-- function we pass it.

let x = c (inc,1).
assert(x == 2).

let y = c (dec,1).
assert(y == 0).
```

Python

- Python supports functions as first-class citizens

```
[lutz$ python3
Python 3.8.2 (default, Jun  8 2021, 11:59:35)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> def inc(i):
[...     return i+1
[...
[>>> foo = inc
[>>> foo(1)
2
>>> 
```

Python

```python
# first-class functions

# define our increment function
def inc(i):
    return i+1

# define our decrement function
def dec(i):
    return i-1

# c expects a function and a value
def c(f,v):
    return f(v)

# we can modify c's behavior depending what kind of
# function we pass it.

x = c (inc,1)
assert(x == 2)

y = c (dec,1)
assert(y == 0)
```

Higher-Order Programming

- Higher-order programming refers to the fact that we take advantage of functions as first-class citizens in our algorithms.

The Lambda Function

- The most well-known feature of higher-order programming is the *lambda* function.
- A lambda function is a function definition without a name.
- In functional-style programming this is often used for functions that are so trivial that they don’t warrant a full function definition

Asteroid:

```
load system "io".

let y = (lambda with x do return x+1) 1.
println y. -- print out the value 2
```

Python:

```
>>> y = (lambda x : x+1) (1)
>>> print(y)
2
>>>
```

The Lambda Function

- Lambda functions are values!

Asteroid:

```
load system "io".

let p = (lambda with x do return x+1).
let y = p 1.
println y. -- print out the value 2
```

Python:

```
>>> p = (lambda x : x+1)
>>> y = p (1)
>>> print(y)
2
>>>
```

- The true power of lambda functions only becomes apparent when combined with other higher-order programming features

The Map Function

- The map function allows you to replace iteration over a list with <u>mapping</u> a function onto the list.
- The map function is a higher-order function since it expects a function as a parameter.

The Map Function

- Asteroid

```
-- compute a list whose elements are incremented
-- by one compared to the input list

let a = [1,2,3].
let b = [].

-- iterate over the list
for e in a do
  b @append(e+1).
end

assert(b == [2,3,4]).
```

```
-- compute a list whose elements are incremented
-- by one compared to the input list

let a = [1,2,3].
let b = [].

-- using map
let b = a @map(lambda with i do return i+1).

assert(b == [2,3,4]).
```

The Map Function

- Python

```python
# compute a list whose elements are incremented
# by one compared to the input list

a = [1,2,3]
b = []

# iterate over the list
for e in a:
  b.append(e+1)

assert(b == [2,3,4])
```

```python
# compute a list whose elements are incremented
# by one compared to the input list

a = [1,2,3]
b = []

# using map
b = map((lambda x : x+1), a)

assert(list(b) == [2,3,4])
```

The Map Function

- One way to think about map is that it applies the given function to each element of the list.

```
[1,2,3] @map(lambda with i do return i+1)
```

```
[
  (lambda with i do return i+1) 1,
  (lambda with i do return i+1) 2,
  (lambda with i do return i+1) 3
]
```

The Map Function

- The lists themselves can consist of structured objects – the supplied function must be able to handle the elements of the list as arguments.
- The return value of the function being mapped can be different from its input values.

```
-- applying map to a list of tuples

let [3,7,11] = [(1,2),(3,4),(5,6)] @map(lambda with (x,y) do return x+y).
```

The Reduce Function

- The reduce function computes a single value from an input list

```
let 6 = [1,2,3] @reduce(lambda with (acc,v) do return acc+v).

let 10 = [1,2,3] @reduce((lambda with (acc,v) do return acc+v), 4).
```

Initial value

The Reduce Function

- The reduce function maps a <u>two-argument</u> function onto a list.
  - First arg acts as accumulator
  - Second arg steps through the elements

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Source: https://realpython.com/python-reduce-function/

The Reduce Function

- Interesting way to implement the factorial operation

Asteroid

```
-- factorial implementation using reduce

let x = 3.
let fact = [1 to x] @reduce(lambda with (acc,j) do return acc*j).

assert(fact == 6).
```

Python

```
from functools import reduce

x = 3

def mul(acc,j):
    return acc*j

fact = reduce(mul, [i for i in range(1,x+1)])

assert(fact == 6)
```

Function Currying

- Function currying is a technique that transforms any multi-argument function into a cascade of single-argument functions.
- The advantage is, this approach allows for <u>partially evaluated functions</u>!

Function Currying

```
function add with (a,b) do
  return a+b.
end
```

```
function addc with a do
  return (lambda with b do return a+b).
end
```

```
let sum = add(1,2).
let sumc = addc 1 2.

assert(sum == 3).
assert(sumc == 3).
```

Arguments to a curried function
are given as a sequence of values!

Function Currying

- Partially evaluated functions!

```
function addc with a do
  return (lambda with b do return a+b).
end
```

```
-- partially evaluated functions!
let partial = addc 1.
let final = partial 2.

assert(final == 3).
```

Function Currying

```
-- demonstration of currying with 3 args

function sum with (a,b,c) do
  return a+b+c.
end

function sumc with a do
  return lambda with b do
    return lambda with c do
      return a+b+c
end

assert( sum(1,2,3) == 6 ).
assert( sumc 1 2 3 == 6 ).
```

Function Currying

- This also works in Python

```
>>> def addc(a):
...     return lambda b : a+b
...
>>> p = addc(1)
>>> p
<function addc.<locals>.<lambda> at 0x102b4baf0>
>>> q = p(2)
>>> q
3
>>> 
```

```
>>> def sumc(a):
...   return lambda b : lambda c : a+b+c
...
>>> sumc(1)(2)(3)
6
>>> 
```

Function Currying

- Why is this interesting?
  - Unique approach to library design – if the library consists of curried functions then the library can be easily extended by partially evaluating those functions
- The libraries for Haskell and ML take this approach

Function Currying

- An example in Python, we assume that we have a library with curried functions…

```
>>> def addc(a):
...   return lambda b : a+b
...
>>> def mulc(a):
...   return lambda b : a*b
...
>>> inc = addc(1)
>>> double = mulc(2)
>>>
>>> inc(1)
2
>>> double(4)
8
>>>
```

Curried library functions

Extending the library with partially evaluated curried functions!

Class Exercise

- Given the Asteroid program on the right do the following:
  - Rewrite inc_list as a recursive function using multi-dispatch.
  - Rewrite inc_list as a function that utilizes the list ‘map’ function to accomplish the computation.

```
function inc_list with input_list do
  let output_list = [].
  for e in input_list do
    output_list @append(e+1).
  end
  return output_list.
end

let l = [1,2,3].
let new_list = inc_list(l).
assert(new_list == [2,3,4]).
```

Assignment

- Assignment #5 – See BrightSpace