EEECS 490 – Lecture 9

Announcements

- Project 2 due Fri 10/6

Agenda

- Lambda Expressions
- Common Functional Patterns

Anonymous Functions

- Names provide an abstraction for an entity that can be used multiple times, but they can be cumbersome if the entity is used in only one place

```
def add_one(x):
    y = 1
    return x + y
```

- This is the case for functions as well

```
def add_one_to_all(values):
    def add_one(x):
        return x + 1
    return map(add_one, values)
```

- Anonymous functions are also called *lambda functions*

Lambda in Scheme

- The `lambda` special form is used to define a function

```
(define (<name> <parameters>) <body>)
→
(define <name> (lambda (<parameters>) <body>))
```

- Nested functions can also be defined with `lambda`

```
(define (make-adder n) (lambda (x) (+ x n)))

> (define add3 (make-adder 3))
> (add3 4)
7
> (add3 5)
8
```

Mutating Non-Local Variables

- In Scheme, there is no distinction between functions defined with **`define`** and those with **`lambda`**

- Nested lambda functions have the full power available to nested functions

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      (- count 1))))

> (define counter (make-counter))
> (counter)
0
> (counter)
1
```

Modify existing binding with set!

Lambda in Python

- A `lambda` expression creates an anonymous function

```
lambda <parameters>: <body expression>
```

- The body must be a single expression, and its value is automatically the return value

```
def make_greater_than(threshold):
    return lambda value: value > threshold

>>> gt3 = make_greater_than(3)
>>> gt3(2)
False
>>> gt3(20)
True
```

Limitations of Python Lambdas

- Python lambdas are lexically scoped and have access to their non-local environment

- However, they are syntactically prohibited from modifying a non-local binding

```
def make_counter():
    count = 0
    return lambda: ???
```

Lambda Expressions in Java

- Lambda expressions in Java create an instance of an anonymous functor-like class

```
static IntPredicate makeGreaterThan(int threshold) {
  return value -> value > threshold;
}
```

Base type of anonymous class inferred from use

Parameters; types are optional

Body can be single expression or a block

```
IntPredicate gt3 = makeGreaterThan(3);
System.out.println(gt3.test(2)); // prints out false
System.out.println(gt3.test(20)); // prints out true
```

Must call method

Limitations of Java Lambdas

- Can only access `final` or "effectively final" variables in enclosing environment

- Effective implementation:

```
static IntPredicate makeGreaterThan(int threshold) {
  return Anonymous(threshold);
}

class Anonymous implements IntPredicate {
  private final int threshold;
  Anonymous(int threshold) {
    this.threshold = threshold;
  }
  public boolean test(int value) {
    return value > threshold;
  }
}
```

Lambda Expressions in C++

- Capture list specifies which variables are captured, and whether they are captured by value or reference

- Can specify default capture as well as specific capture type for individual variables

```
auto make_greater_than(int threshold) {
  return [=](int value) {
    return value > threshold;
  };
}
```

Type deduction used to infer type of lambda

Capture non-locals used in lambda by value

```
auto gt3 = make_greater_than(3);
cout << gt3(2) << endl;
cout << gt3(20) << endl;
```

Capture-Free Lambdas

- Lambdas that don't capture anything are equivalent to top-level functions

- They can even bind to function pointers

```
int max_element(int *array, size_t size,
                bool (*less)(int, int));

int main() {
  int array[4] = { 3, 1, 4, 2, 5 };
  cout << max_element(array, 5,
                      [](int a, int b) {
                        return a > b;
                      })
       << endl;
}
```

Lambdas and Functors

- Capturing lambda equivalent to instance of anonymous class

```
void foo(int x, int y) {
  auto fn = [=x, &y](int z) { y = x + z; };
  auto fn2 = Anon(x, y);  // equivalent
}

class Anon {
  const int x;
  int &y;
public:
  Anon(int xin, int &yin) : x(xin), y(yin) {}
  void operator()(int z) {
    y = x + z;
  }
};
```

Variables captured by value are const by default

Lifetime of Captured Objects

- Capturing a variable by reference does not extend its lifetime
  - RAII requires that an automatic variable be destroyed upon exit from its enclosing scope

- Programmer needs to ensure that a capturing lambda is not used after captured objects die

❌
```
auto make_counter() {
  int count = 0;
  return [&]() {
    return count++;
  };
}
```

✅
```
auto make_counter() {
  int count = 0;
  return [=]() mutable {
    return count++;
  };
}
```

Allows variable captured by value to be modified

Break Time!

- We'll start again in five minutes.

Map

- The *map* pattern applies a function to each element in a sequence, producing a new sequence consisting of the results

```
(define (map fn items)
  (if (null? items)
      '()
      (cons (fn (car items))
            (map fn (cdr items)))))

> (map (lambda (x) (+ x 1)) '(1 2 3 4))
(2 3 4 5)
```

Reduce

- The *reduce* pattern applies a function to a pair of items from a sequence, then to the result of that and the next item, and so on until the sequence is exhausted

- Can be right or left associative

```
(define (reduce-right fn items)
  (if (null? (cdr items))
      (car items)
      (fn (car items)
          (reduce-right fn (cdr items)))))

> (reduce-right (lambda (x y) (+ x y)) '(1 2 3))
6
> (reduce-right (lambda (x y) (if (> x y) x y))
                '(2 3 4 1))
4
```

Assumes at least one item in list

Filter

- The *filter* pattern applies a predicate to the items in a sequence, producing a new sequence that only includes the items that test true

```
(define (filter pred items)
  (if (null? items)
      items
      (if (pred (car items))
          (cons (car items)
                (filter pred (cdr items)))
          (filter pred (cdr items)))))

> (filter (lambda (x) (> x 0)) '(1 -3 4 0))
(1 4)
> (filter (lambda (x) (even? x)) '(1 -3 4 0))
(4 0)
```

Map, Reduce, Filter in Python

- Python has built-in map, reduce, and filter
- Reduce located in `functools` module
- Result of `map()` and `filter()` are separate iterator types

```
>>> from functools import reduce
>>> map(lambda x: x + 1, [1, 2, 3, 4])
<map object at 0x10b438390>
>>> list(map(lambda x: x + 1, [1, 2, 3, 4]))
[2, 3, 4, 5]
>>> list(filter(lambda x: x > 0, [-1, 2, 0, 4]))
[2, 4]
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4])
10
```

Any and Every

- The *any* and *every* patterns are higher-order generalizations of *or* and *and*

```
(define (any pred items)
  (if (null? items)
      #f
      (let ((result (pred (car items))))
        (if result
            result
            (any pred (cdr items))))))
```

Evaluate to first result of pred that is true

```
> (any (lambda (x) (< x 0)) '(1 2 3 4))
#f
> (any (lambda (x) (< x 0)) '(1 -2 -3 4))
#t
```

Compose

- Applying a function to the result of another function is a common operation

- Can define the composition of two functions

```
(define (compose f g)
  (lambda (x) (f (g x))))

> (map (compose (lambda (x) (+ x 1))
                (lambda (x) (* 3 x)))
       '(3 5 7))
(10 16 22)
```