EECS 490 – Lecture 7

Recursion and Higher-Order Functions

9/26/17

Announcements

- Homework 2 due on Friday

- Project 2 due Fri 10/6

Review: Call by Result

- Argument must be l-value
- Parameter is a new variable with its own storage
- Parameter is **not** initialized with argument value
- Upon return of the function, parameter value is copied to argument object
- Can only be used for output

```
void foo(result int x) {
  x = 3;
  ...
  x++;    // x is now 4
}

int y = 5;
foo(y);   // y is now 4
```

Review: Call by Value-Result

- Combination of call by value and call by result
- Argument must be l-value
- Parameter is a new variable with storage, initialized with argument value
- Upon return, value of parameter is copied to argument object

```
int foo(v/r int x, v/r int y) {
  x++;
  return x - y;
}

int z = 3;
print(foo(z, z));    // prints 1
```

Again, not C++! Final value of `z` depends on whether it is copied from first or second parameter in the given language

Call by Name

- Any expression provided as argument
- Parameter name is replaced by argument expression everywhere in the body
- Expression computed whenever it is encountered in body

```
void foo(name int x) {
  print(x);  // becomes print(++y)
  print(x);  // becomes print(++y)
}

int y = 3;
foo(++y);  // prints 4, then 5; y is now 5
```

!C++; Mutating expressions should not be passed by name, since behavior would depend on implementation details

Thunks

- In call by name, expression must be computed in its own environment

```
void bar(name int x) {
  int y = 3;
  print(x + y);  // becomes print(y + 1 + y)
}

int y = 1;
bar(y + 1);      // should print 5, not 7
```

- This is accomplished with a *thunk*, a compiler-generated local function that packages the expression with its environment

Python is Call by Value

- Call by value is most common mode, followed by call by reference
- Python and Java are not call by reference
  - They combine call by value with reference semantics
  - This is sometimes called "call by object reference"

```
def swap(x, y):
    tmp = x
    x = y
    y = tmp

>>> x, y = 1, 2
>>> swap(x, y)
>>> x, y
(1, 2)
```

x and y are new variables with their own storage

Agenda

- Recursion
- Function Objects
- Functions as Parameters
- Nested Functions

Overview of Recursion

- A function is recursive if it calls itself directly or indirectly
- A recursive computation has
  - Base cases: cases that can be computed directly without recursion
  - Recursive cases: a case that can be computed from the solution to a "smaller" case; the smaller case is solved with a recursive call
- Recursion can be used for repetition instead of iteration

Activation Records

- Recursion works on a machine since every function invocation gets its own activation record

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Implicit Data in Activation Records

- An activation record includes implicit data needed by the function invocation
  - Storage for temporary values
  - Address where to place the return value
  - Address of caller's code and activation record

- The set of implicit items can be determined statically

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Space Usage of Factorial

- Computation of `factorial(n)` requires *n + 1* invocations to be active at the same time

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

- Compare to iterative version:

```
def factorial_iter(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

Alternate Definition of Factorial

- We can define another recursive version that:
  - Does no computation after the recursive call
  - Directly returns the result of the recursive call

```
def factorial_tail(n, result = 1):
    if n == 0:
        return result
    return factorial_tail(n - 1, n * result)
```

Tail-Call Optimization

- A call is a *tail call* if its caller directly returns the result without performing additional computation
- Tail-call optimization reuses the space for the caller's activation record for that of the tail call
- Some implicit data is also reused for the tail call:
  - Address where to place return value
  - Address of caller's code and activation record

```
def factorial_tail(n, result = 1):
    if n == 0:
        return result
    return factorial_tail(n - 1, n * result)

def foo():
    print(factorial(4))
```

Tall-Call Optimization Failures

- Implicit computation, such as destructors, can prevent optimization

```
int sum(vector<int> values, int index,
        int partial_result = 0) {
  if (values.size() == index) { return 0; }
  return sum(values, index + 1,
             partial_result + values[index])
}
```

- Nested function definitions can prevent optimization

Function Objects and State

- A *function object* (also called a *functor*) is an object that isn't a function but provides the same interface
- Allowing the function-call operator to be overridden enables function objects to be defined
- Function objects can have state that is associated with an instance of the functor
  - State shared among all invocations of the same instance
  - Different than top-level functions, which only have state that is associated with a single invocation or with all invocations of the function

Function Objects in C++

- Functors can be written by defining a class that overrides the `operator()` member function

```
class Counter {
public:
  Counter : count(0) {}
  int operator()() {
    return count++;
  }
private:
  int count;
};
```

Can have parameters, just like functions

```
Counter counter1, counter2;
cout << counter1() << endl; // prints 0
cout << counter1() << endl; // prints 1
cout << counter1() << endl; // prints 2
cout << counter2() << endl; // prints 0
cout << counter2() << endl; // prints 1
cout << counter1() << endl; // prints 3
```

Function Objects in Python

- Functors override the `__call__` special method

```
class Counter:
    def __init__(self):
        self.count = 0
    def __call__(self):
        self.count += 1
        return self.count - 1
```

More parameters can go here

```
counter1 = Counter()
counter2 = Counter()
print(counter1()) # prints 0
print(counter1()) # prints 1
print(counter1()) # prints 2
print(counter2()) # prints 0
print(counter2()) # prints 1
print(counter1()) # prints 3
```

"Function Objects" in Java

- Java does not have operator overloading
- Instead, "function objects" implement an interface with a conventional name for the lone method in the interface

```
class Counter implements Supplier<Integer> {
  private int count = 0;
  public Integer get() {
    return count++;
  }
}
```

```
Supplier<Integer> counter1 = new Counter();
Supplier<Integer> counter2 = new Counter();
System.out.println(counter1.get()); // prints 0
System.out.println(counter1.get()); // prints 1
System.out.println(counter1.get()); // prints 2
System.out.println(counter2.get()); // prints 0
System.out.println(counter2.get()); // prints 1
System.out.println(counter1.get()); // prints 3
```

Break Time!

- We'll start again in five minutes.

Function Pointers

- C and C++ allow top-level functions to be passed by pointer

```
void apply(int *A, size_t size, int (*f)(int)) {
  for (; size > 0; --size, ++A)
    *A = f(*A);
}

int add_one(int x) {
  return x + 1;
}

int main() {
  int A[5] = { 1, 2, 3, 4, 5 };
  apply(A, 5, add_one);
  cout << A[0] << ", " << A[1] << ", " << A[2]
       << ", " << A[3] << ", " << A[4] << endl;
}
```

Automatically converted to function pointer

Environment of Use

- A function passed as a parameter has three environments that can be associated with it
  - The environment where it was defined
  - The environment where it was referenced
  - The environment where it was called
- Scope policy determines which names are visible in the function
  - Static/lexical scope: names visible at the definition point
  - Dynamic scope: names visible at the point of use
- In dynamic scope, point of use can be where a function is referenced or where it is called

Binding Policy

- *Shallow binding*: non-local environment is environment from where a function is called
- *Deep binding*: non-local environment is environment from where a function is referenced

```
int foo(int (*bar)()) {
  int x = 3;
  return bar();
}

int baz() {
  return x;
}

int main() {
  int x = 4;
  print(foo(baz));
}
```

Non-local environment in shallow binding

Non-local environment in deep binding

C-like code with dynamic scope.

Nested Functions and Closures

- The ability to create a function from within another function is a key feature of functional programming
- Static scope requires that the newly created function have access to its definition environment
- A *closure* is the combination of a function and its enclosing environment
- Variables from the enclosing environment that are used in the function are *captured* by the closure

Nested Functions and State

- A closure encompasses state that can be accessed by the newly created function

```
def make_greater_than(threshold):
    def greater_than(x):
        return x > threshold
    return greater_than
```

threshold captured from non-local environment

```
>>> gt3 = make_greater_than(3)
>>> gt30 = make_greater_than(30)
>>> gt3(2)
False
>>> gt3(20)
True
>>> gt30(20), gt30(200)
(False, True)
```

Modifying Non-Local State

- Languages may allow non-local variables to be modified

```
def make_account(balance):
    def deposit(amount):
        nonlocal balance
        balance += amount
        return balance
    def withdraw(amount):
        nonlocal balance
        if 0 <= amount <= balance:
            balance -= amount
            return amount
        else:
            return 0
    return deposit, withdraw
```

```
>>> deposit, withdraw = \
        make_account(100)
>>> withdraw(10)
10
>>> deposit(0)
90
>>> withdraw(20)
20
>>> deposit(0)
70
>>> deposit(10)
80
>>> withdraw(100)
0
>>> deposit(0)
80
```

We will come back to data abstraction using functions later.

Decorators

- A common pattern in Python is to transform a function or class by applying a higher-order function to it, called a *decorator*

- Standard syntax for decorating functions:

```
@<decorator>
def <name>(<parameters>):
    <body>
```

- Mostly equivalent to:

```
def <name>(<parameters>):
    <body>

<name> = <decorator>(<name>)
```

Trace Example

- Example: decorator that traces function calls

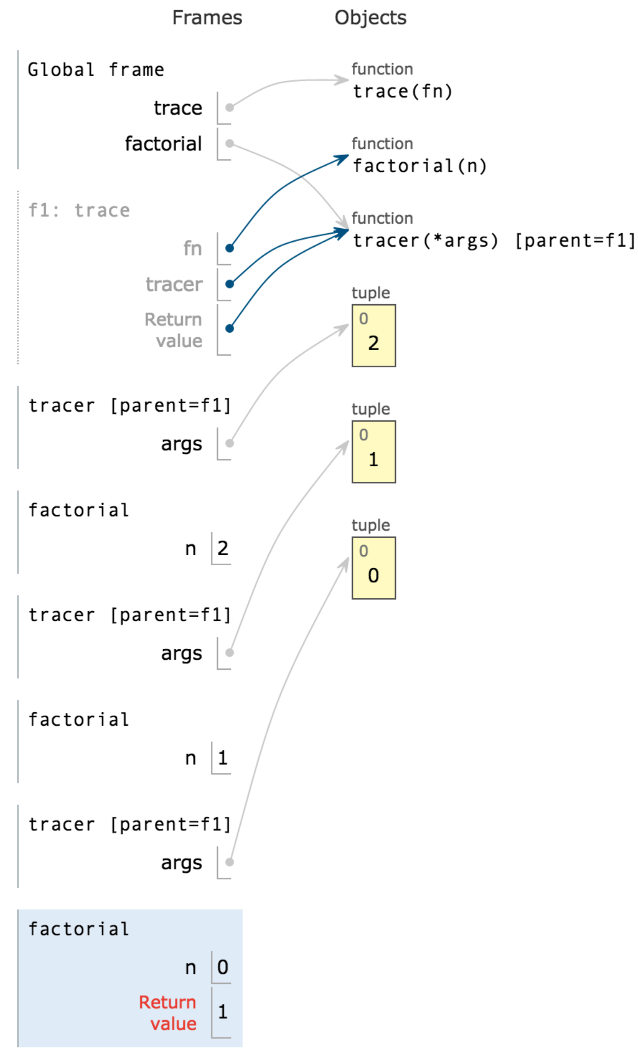
```
def trace(fn):
    def tracer(*args):
        strs = (str(arg) for arg in args)
        print('{}({})'.format(fn.__name__,
                              ', '.join(strs)))
        return fn(*args)
    return tracer

@trace
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```
>>> factorial(5)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)
120
```

Mutual Recursion

- A decorated recursive function results in *mutual recursion* where multiple functions make recursive calls indirectly through each other

```
>>> factorial(2)
factorial(2)
factorial(1)
factorial(0)
2
```



This example on Python Tutor: https://goo.gl/issW90