CS 315 – Programming Languages
Subprograms

Subprograms (aka. functions)

- Definition
  - header: name, parameters (formals), return type
  - body: code
  - E.g.: `int triple (int x) { return 3 * x; }`
- Call
  - name, parameters (actuals)
  - E.g.: `triple(3)`

Design issues about subprograms

- Parameter passing semantics/syntax (already discussed)
- Functions as arguments/return types
- Nested function definitions
- Closures: nested functions that reference variables form the enclosing function scope
- Generic subprograms
- Co-routines

Parameters

- Positional (Java, C++)
  - order matters (one-to-one mapping of actuals to formals)
  - optional parameters appear at the end
- Keyword-based (swift, ellielang)
  - order does not matter
  - when mixed, keyword-based parameters appear after positional ones
- Variable number of parameters

E.g.: Python is very powerful in terms of parameter passing

- Positional

```
def sum(x, y, z):
    return x + y + z
u = sum(a, b, c)
```

- Positional with defaults

```
def sum(x, y, z=3):
  return x + y + z
u = sum(a, b)
u = sum(a, b, c)
```

- Keyword-based

```
def sum(x, y, z=3):
  return x + y + z
```

```
u = sum(y=a, x=b)
u = sum(y=a, z=c, x=b)
```

- Variable number of args

```
def sum(x, y, *z):
    s = x + y
    for e in z:
        s = s + e
    return s
u = sum(a, b)
u = sum(a, b, c, 3, 5)

def sum(x, y, **z):
    s = x + y
    for k, e in z.iteritems():
        s = s + e
    return s
u = sum(a, b)
u = sum(a, b, i=3, j=5, k=6)
u = sum(x=a, y=b, i=3, j=5, k=6)

def sum(x, y, *z, **w):
    s = x + y
    for e in z:
        s = s + e
    for k, e in w.iteritems():
        s = s + e
    return s
u = sum(a, b)
u = sum(a, b, c, d, i=3, j=5, k=6)

# Python also supports unpacking via *
v = [3, 5, 6]
u = sum(*v) # equivalent to u = sum(3, 5, 6)
v = {'i': 3, 'j': 5, 'k': 6}
u = sum(**v) # equivalent to u = sum(i=3, j=5, k=6)
```

<u>Models of parameter passing</u>

Semantics

- in mode — actuals are copied into formals during the initiation of the call
- out mode — formals are copied into actuals during the return of the call
- inout mode — actuals are copied into formals during the initiation of the call and formals are copied into actuals during the return of the call

Common implementations

- pass-by-value: in-mode
- pass-by-result: out-mode
- pass-by-value-result: in-out mode
- pass-by-reference: a variation of in-out mode where there is no copy, but the formal references the actual, which means they are aliases (when one changes the other does too)

E.g.: C++

```
void foo(int x, int * y, int & z) { … }

int a, b, c;
foo (a, &b, c);
// first parameter is passed by value
// second parameter, which is a pointer, is also passed by value
//   but it can be used to implement inout semantics for b
// third parameter is passed by reference
```

```
void swap1(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
swap1(c, d); // does not swap
```

```
void swap2(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
swap2(&c, &d);
```

```
void swap3(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
swap3(c, d);
```

Q: When does pass-by-value-result and pass-by-reference give different results?

Assume `inout` specifies pass-by-value-result and `ref` specifies pass-by-reference.

```
void foo(inout int a, inout int b)
{
     a++;
     b++;
}
x = 2;
foo(x, x);
// x is 3
```

```
void foo(ref int a, ref int b)
{
     a++;
     b++;
}
x = 2;
foo(x, x);
// x is 4
```

Pass-by-reference has performance advantages as it does not need to copy data. However, it results in creation of aliases (as in the above example), which might be hard to manage sometimes.

Passing functions as parameters

For dynamic languages, this is easy / no type checking at compile-time. E.g.: Python

```
def map(op, items):
     res = []
     for item in items:
          res.append(op(item))
     return res

def square(x):
     return x * x
print map(square, [1, 2, 3]) # [1, 4, 9]

triple = lambda x : 3*x
print map(triple, [1, 2, 3]) # [3, 6, 9]
```

For languages like C/C++ you need type checking. E.g.: C++

```
int square (int x)
{
     return x * x;
}
vector<int> map(int (*op)(int a), vector<int> const & list)
{
     vector<int> res;
     for (int val : list)
          res.push_back((*op)(val));
     return res;
}
vector<int> list = {1, 2, 3, 4};
vector<int> res = map(&square, list);
```

Closures

Subprograms together with their referencing environment, including variables referenced from the enclosing scope.

Example:

Python:

```
def make_adder(x):
     def adder(y):
          return x + y # here x is from make_adder
     return adder
adder = make_adder(3)
print adder(5) # 8
```

C++:

```
function<int(int)> make_adder(int x) {
     auto adder = [=] (int y) { return x + y; };
     return adder;
}
auto adder = make_adder(3);
cout << adder(5); // 8
```

Java:

```
Function<Integer, Integer> make_adder(Integer x) {
  Function<Integer,Integer> adder = new Function<Integer,Integer>() {
     @Override
     public Integer apply(Integer y) { return x + y; }
  };
  return adder;
}
Function<Integer, Integer> add3 = make_adder(3);
System.out.println(add3.apply(5)); // 8
```

It is important to note that the variables used within the closure may be from the referencing environment, in which case, such variables may be changed elsewhere, impacting the closure.

```
def make_appender(post_list):
    def appender(pre_list):
        pre_list.extend(post_list)
    return appender

X = [4, 5]
Y = [1, 2]
Z = [1, 2]
appender = make_appender(X)
appender(Y)
print Y # [1, 2, 4, 5]
X.insert(0, 3) # insert 3 to front
appender(Z)
print Z # [1, 2, 3, 4, 5]

# Question: What if I replace
# X.insert(0, 3) with X = [3, 4, 5]
```

```
import copy
def make_appender(post_list):
    pl_copy = copy.deepcopy(post_list)
    def appender(pre_list):
        pre_list.extend(pl_copy)
    return appender

X = [4, 5]
Y = [1, 2]
Z = [1, 2]
appender = make_appender(X)
appender(Y)
print Y # [1, 2, 4, 5]
X.insert(0, 3) # insert 3 to front
appender(Z)
print Z # [1, 2, 4, 5]
```

X will be assigned to a new list.

However, the make_appender( ) function will now have a reference to the old X.

```
# Answer: Z would be [1, 2, 4, 5]
```

We can also modify the variable that comes from the enclosing function form within the closure. E.g.:

```
def make_counter():
     next_value = 0
     def return_next_value():
          value = next_value
          make_counter.next_value = val + 1
          # Python 2.0 requires fully qualifying nonlocal variables
          # that are being mutated (thus the make_counter.)
          return value
     return return_next_value

c1 = make_counter()
c2 = make_counter()

print c1() # 0
print c1() # 1
print c1() # 2
print c2() # 0
print c2() # 1
```

VERY TRICKY!

In C++, one needs to be careful about such local variables (which are on the stack and go out of the scope after the function returns):

```
#include <functional>
using namespace std;

function<int (void)> make_counter() {
     int next_value = 0;
     auto return_next_value = [&] () { // [&] captures by reference
          int value = next_value;
          next_value++;
          return value;
     };
     return return_next_value;
}

auto c1 = make_counter();
auto c2 = make_counter();
c1(); c1(); c1(); c2(); c2();
```

The above program is buggy. The reason is that, `next_value` is stored on the stack. When the function returns, the value is gone (its lifetime is over). To fix this, we need to use heap allocation.

```
function<int (void)> make_counter() {
     shared_ptr<int> next_value = new int(0);
     auto return_next_value = [=] () { // [=] captures by value
                                        // that is, pointer is copied
          int value = *next_value;
          (*next_value)++;
          return value;
     }
     return return_next_value;
```

```
}
```

Generic sub-programs

Ability to write subprograms that work with more than one type.

C++

```
template <class T>
T max(T first, T second) {
    return first > second ? first : second;
}
```

What to do without templates? C has macros:

```
#define max(a, b) (((a)>(b))?(a):(b))
```

Why do we have so many parentheses? Because macros use text substitution and if `a` and `b` above are expressions, they may mess up the operator precedence.

Still, macros are not safe. E.g.: Consider `max(x++, y)`, which translates into

```
(((x++)>(y))?(x++):(y))
```

In some cases, this would result in incrementing `x` twice.

Java

class name <T> {…}
<T> return_type metod_name(...) {...}

A few important differences:

- type parameters must be classes, not primitive types
- Java uses a single method/class via type erasure (ArrayList<Integer> and ArrayList<Float> are the same type at run-time (ArrayList<Object>), but in C++ these would be different types)
- Java supports restrictions

```
public static <T> T max(T[] list) { … }
public static <T extens Comparable<T>> T max(T[] list) { … }
```

Wildcards:

```
public void drawAll(ArrayList<? extends Shape> things) { … }
public <T extends Shape> void drawAll(ArrayList<T> things) { … }
```

The second alternative is more powerful (can enforce same or different types):

```
public bool compare(List<?> a, List<?> b) { … }
public <T1, T2> bool compare(List<T1> a, List<T2> b) { … }
public <T> bool compare(List<T> a, List<T> b) { … }
```

Co-routines

Co-routines are multi-entry, multi-exit functions. A specialized form of co-routines that is more popular is called *generators*.

E.g.: Python supports generators.

Consider the problem of printing all 2 combinations for numbers in the range [0, n):

```
# printing 2 combinations
def print2Combs(n):
    for i in xrange(0, n):
        for j in xrange(i+1, n):
            print (i,j)

print2Combs(5)
print
```

Consider a similar problem, where the goal is to iterate, rather than print.

```
class CombIter:
    def __init__(self, n):
        self.n = n
    def next(self):
        self.j = self.j + 1
        if (self.j==self.n):
            self.i = self.i + 1
            self.j = self.i + 1
        if (self.i==self.n-1):
            raise StopIteration
        return (self.i, self.j)
    def __iter__(self):
        self.i = 0
        self.j = 0
        return self

for p in CombIter(5):
    print p # or do whatever you want to do with p
```

There is a much easier way of doing this with generators:

```
def gen2Combs(n):
    for i in xrange(0, n):
        for j in xrange(i+1, n):
            yield (i,j)

for p in gen2Combs(5):
     print p # or do whatever you want to do with p
```

The `yield` statement results in returning a value, but the next time the function is called, it continues from the exact place where it yielded.