Programming Language Concepts
Subroutines

Janyl Jumadinova
23 February, 2017
Arguments, Parameters

Arguments are also called "actual parameters" (as opposed to "formal parameters" in the function definition).

```plaintext
x = f(3, a, b+c);  // Arguments
int f(int x, int y, int z){  // Parameters
  ...
```

2/20
Arguments are also called “actual parameters” (as opposed to “formal parameters” in the function definition)

```c
int f(int x, int y, int z){
...
```

```c
x = f(3,a,b+c);
...```

Arguments, Parameters
Examples of Function Calls (Python)

```python
def f(a, b, c):
    ... return 100*a + 10*b + c
```
Examples of Function Calls (Python)

```
def f(a=1, b=1, c=1):
    ... return 100*a+10*b+c
```

- Default Parameter Values
Examples of Function Calls (Python)

```python
def f(a=1, b=1, c=1):
    ... return 100*a + 10*b + c
```

- Default Parameter Values
- Named Arguments
Examples of Function Calls (Haskell)

Prelude> let f a b c = 100 * a + 10 * b + c
Prelude> f 10 20 30
1230

Prelude> let g = f 1
Prelude> g 20 30
330

Prelude> let h = g 1
Prelude> h 30
140

- g is a function obtained by giving parameter “a” the value “1” in function f. This is called “currying”.

- More currying; h x is equivalent to g 1 x, which is equivalent to f 1 1 x
Parameter Evaluation

“Applicative Order” evaluation:
arguments evaluated before the function call. “Eager” evaluation.

```c
int slow(int n){
    /* count to n^2 */
    return count;
}
int main(){
    int x;
    x = f(10, slow(1000000));
    printf("%d\n", x);
}
int f(int a, int b){
    return a+1;
}
```

```
time ./a.out
a = 11
real 0m20.131s
user 0m20.126s
sys 0m0.000s
```
Parameter Evaluation

“Normal Order” evaluation:
arguments are not evaluated until they are needed (possibly never)

File `lazy.hs`:

```hs
slow 0 = 0
slow n = 1 + slow (n-1)
f a b = a + 1
```

In ghci, try:

```
Prelude> :l lazy
[1 of 1] Compiling Main ...
Ok, modules loaded: Main.
*Main> f 10 (slow 10000000)
11
*Main> f (slow 10000000) 10
10000001
```
“Lazy” evaluation:
- arguments are evaluated at most once (possibly never).
- Even though we used a Haskell example to illustrate “normal order”, it is more accurate to call Haskell’s evaluation order “lazy.”
- In normal order, a parameter could be evaluated more than once (i.e., evaluated each time it appears in the function).
In languages that support “first-class functions,” a function may be a return value from another function; a function may be assigned to a variable. This raises some issues regarding scope.
Closures

JavaScript example: [http://goo.gl/kzUCes](http://goo.gl/kzUCes)

```javascript
function f(name) {
  var x = "hi there";
  function g() {
    return x + " " + name;
  }
  return g;
}
var k = f("bob");
```

Function $f$ returns the function $g$. Therefore, variable $k$ is assigned a function. Once $f$ is done, how will $k$ (i.e., $g$) know the values of $x$ and $name$?
Closures

One solution (NOT the one used by JavaScript!):

use the most recently-declared values of variables “name” and “x”.
– This is called “shallow binding.” Common in dynamically-scoped languages.
Closures

One solution (NOT the one used by JavaScript!):
use the most recently-declared values of variables “name” and “x”.
– This is called “shallow binding.” Common in dynamically-scoped languages.

Another solution (used by JavaScript and most other statically-scoped languages):
bind the variables that are in the environment where the function is defined.
– This is an illustration of “deep binding” and the combination of the function and its defining environment is called a closure.
Closures: Another example (C#)

```csharp
static void Main(string[] args) {
    Func<int,int> g = returnFunc(7);
    Console.WriteLine(g(9));
}
static Func<int,int> returnFunc(int x) {
    Func<int,int> g =
    delegate(int z) { int y = 10; return x+y+z; };
    return g;
}

See Closurer.cs in the repo.
```
Exceptions

See Exc.java in the repo.
Coroutines

- A coroutine is a function that can be suspended and resumed.
- Several coroutines can be active at once, transferring control back and forth between them.
Coroutines are supported in languages such as Simula and Modula-2; useful for writing discrete event simulation code.
Generators in Python

Many other languages have features that allow the implementation of coroutines (even if they are not built in to the language).
– Python has generator functions:

```python
>>> def gen99():
...   for i in range(100):
...     yield i # NOTE: not return i

>>> a = gen99() # call the function just once
>>> next(a)
0
>>> next(a)
1
```
Generators in Python

```python
>>> next(a)
2
>>> for i in range(10):
...     print next(a),
...
3 4 5 6 7 8 9 10 11 12
>>> for i in range(10):
...     print next(a),
...
13 14 15 16 17 18 19 20 21 22
```
Several generators can be active at the same time; see sample program gen.py in the shared repository.

This isn’t precisely a coroutine example (we don’t have “call” and “response” directly transferring back and forth).

See https://docs.python.org/3/library/asyncio-task.html ("Tasks and coroutines", Python 3.5 documentation) intersection.
Functions: Summary

We have considered the following topics:

- Parameter passing (e.g., pass by value, pass by reference)
- Special syntax (default values, named parameters)
- Mechanisms for function calls (activation record stack, static and dynamic pointers, calling sequences)
- Parameter evaluation (applicative, normal, lazy)
- Closures
- Exceptions
- Coroutines
Functions: Summary

We didn’t cover:

- Generics (you have seen a lot of this in CMPSC 112)
- Events (we will revisit this topic in Chapter 13).