Subprograms: Lambdas, Closures, Generators, and Coroutines

Programming Languages

William Killian

Millersville University

Outline

- Lambda Functions
- Closures
- Re-entrant Subprograms
  - Generators
  - Coroutines

Lambda Functions

Lambda Functions

- **Also called**
  - function literal
  - lambda abstraction
  - anonymous function
  - lambda expression
- Originate from *Alonzo Church* from his invention of Lambda Calculus around 1936 (all functions were anonymous)

*In programming languages since **1958***

Lambda Function Tour

Lisp

```
(lambda () sexpr )
(lambda (x y) (+ x y))
```

JavaScript

```
() => expr
() => { stmts… }
(x, y) => x + y
function(x, y) { return x + y; }
```

Lambda Function Tour

OCaml

```
fun () -> ...
fun x y -> x + y
```

Java

```
() -> expr
() -> { stmts… }
(x, y) -> x + y
(x, y) -> { return x + y; }
(double x, double y) -> { return x + y; }
```

Lambda Function Tour

Python

```
lambda: expr
lambda x, y: x + y
```

Swift

```
{ () in expr }
{ () in stmts… }
{ x, y in x + y }
{ (x: int, y: int) -> int in return x + y }
{ $0 + $1 }
```

Lambda Function Tour

C++

```
[]() { stmts… }
[](auto x, auto y) { return x + y; }
[](int x, int y) -> int { return x + y; }
```

Ruby

```
lambda { expr }
lambda { stmts… }
lambda { |x, y| x + y }
-> x, y { x + y }
```

Closures

Closures

- Closures need not have a name (but they can)
- **Key difference:** closures access *non-local variables*
- Most often disguised as a lambda with no syntactical difference
  - Lisp
  - Java
  - Javascript
  - OCaml
  - Python
  - Swift
  - Ruby
- Exception: C++

Closure Example

JavaScript

```
x => y => x + y
x => { return y => x + y }
                       ^
```

OCaml

```
let multiply n list =
  List.map (fun x -> n * x) list
                         ^
```

Closure Example

C++

```
std::vector<int>
multiply (int n, std::vector<int> list) {
  std::for_each (list, [n] (int& value) {
    return value * n; //^ explicit copy of n
  });
  return list;
}
```

Closure Example

C++

```
std::vector<int>
multiply (int n, std::vector<int> list) {
  std::for_each (list, [=] (int& value) {
    return value * n; //^ all non-locals are
  });                 //  copied and stored
  return list;
}
```

Closure Example

C++

```
std::vector<int>
multiply (int n, std::vector<int> list) {
  std::for_each (list, [&] (int& value) {
    return value * n; //^ all non-locals are
  });                 //  passed by reference
  return list;
}
```

Re-Entrant Subprograms

Function Definition

A function is a subprogram that…

- Has exactly one entry point
- May have one or more exit points

Function Definition

A function is a subprogram that…

- Has exactly one entry point
  - The beginning of the function!
- May have one or more exit points
  - Return statements throughout
  - Exceptional control flow (coming up)

- What if we could re-enter a function?

Generators

- Generators are the simplest type of re-entrant subprogram.
- Generators can give us different values each invocation time
- Values are not computed when the sequence is created, but when they are asked for!

Generators

Goal

- Don’t want to exit the subprogram, simply “pause” it in some way

- Achieved by **not** using `return`

- Define a new control flow keyword!

Generators

- `yield` values
- Yield “pauses” execution of the subprogram.
- When we call the subprogram we resume from where we left off!

Python:

```
def first_three():
    yield 1
    yield 2
    yield 3
```

Generators

```
def ones():
    while True:
        yield 1

def natural():
    x = 0
    while True:
        x += 1
        yield x
```

```
gen = natural()
next(gen)
next(gen)
next(gen)
next(gen)
```

Generators

```
# Exercise
def fibonacci():
```

Recursive Generators

```
from os import listdir
from os.path import isfile, join, exists

def print_files(path):
  for file in listdir(path):
    full_path = join(path, file)
    if exists(full_path):
      if isfile(full_path):
        yield full_path
      else:
        yield from print_files(full_path)
```

Coroutines

- A generalization of all subprograms
  - Execution of a subprogram can be **paused** or **resumed**
  - Often used for multi-tasking and concurrent programming
- Subroutines
  - Called once, returned once
  - Never pauses exection
- Generators
  - Called multiple times, returns values multiple times
  - **Pauses** execution immediately after yielding
- Coroutines
  - Called multiple times, returns values multiple times
  - Execution can continue after yielding a value

Coroutines: Language Support

- C++ (since C++20)
- C#
- D
- F#
- Go
- JavaScript
- Julia
- Lua
- PHP
- Prolog
- Python
- Ruby
- Rust
- Scheme (lisp-like)

Coroutines

- Subprograms that both produce and consume values which are “yielded” are called *coroutines*

- can also consume values using the **yield** expression (different from the **yield** statement!)

Coroutines Example

```
def match(pattern):
    print('Looking for ' + pattern)
    try:
        while True:
            s = (yield)
            if pattern in s:
                print(s)
    except GeneratorExit:
        print('Done.')
```

Coroutines Example

```
>>> matcher = match('hello')
>>> next(matcher)
    Looking for hello
>>> matcher.send('hello there')
    hello there
>>> matcher.send('goodbye now')
>>> matcher.send('Othello is a great play')
    Othello is a great play
>>> matcher.close()
    Done.
```