Coroutines in Lua

Coroutines

- An unconventional, but quite powerful control mechanism
- Well known as an abstract concept, but with several variations
- Variations with big differences

Kinds of Coroutines

- Symmetric or asymmetric
- *Stackful*
- First-class values

Symmetric and Asymmetric Coroutines

- Symmetric coroutines: one primitive for transfering control
  - Typically called `transfer`
- Asymmetric coroutines: two primitives for transferring control
  - Typically called `resume` and `yield`

```
resume/yield  ↔  call/return
transfer      ↔  goto
```

Stackful Coroutines

- Non-stackful coroutines can be suspended only inside the body of the original function
  - Original concept (co-routine x sub-routine)
- Stackful coroutines can be suspended while calling other functions
  - As implemented in Modula
  - Similar to cooperative multithreading

First-class Coroutines

- Coroutines can be represented by first-class values
  - can be resumed anywhere in a program
- Restricted forms of coroutines are not first class
  - e.g., generators in CLU and other languages

Full Coroutines

- A *full coroutine* is a stackful, first-class coroutine
- For full coroutines, symmetric and asymmetric control are equivalent
  - you can implement one with the other
  - just like goto x call/return
- Full coroutines are equivalent to one-shot continuations
  - you can implement `call/1cc` with them

Coroutines in Lua

- Full, asymmetric coroutines
- Full coroutines present one-shot continuations in a format that is more familiar to conventional programmers
  - similar to multithreading
- Full coroutines allow a simple and efficient implementation
  - as compared with one-shot continuations

Asymmetric coroutines

- Asymmetric and symmetric coroutines are equivalent
- Not when there are different kinds of contexts
  - integration with C
- How to do a `transfer` with C activation records in the stack?
- `resume` fits naturally in the C API

Coroutines: First Example

```
co = coroutine.wrap(function (x)
       print(x)
       coroutine.yield()
       print(2*x)
     end)

co(20)                  --> 20
co()                    --> 40
co()
  --> error: cannot resume dead coroutine
```

Coroutines: exchanging values

```
co = coroutine.wrap(function (x)
       x = coroutine.yield(2*x)
       return 3*x
     end)

print(co(20))         --> 40
print(co(2))          --> 6
co()
  --> error: cannot resume dead coroutine
```

Producer - Consumer

```
function produce ()
  while true do
    local x = io.read()
    send(x)
  end
end
```

```
function consume ()
  while true do
    local x = receive()
    print(x)
  end
end
```

```
send = coroutine.yield
receive = coroutine.wrap(produce)

consume()
```

Coroutines and Iterators

```
function permgen (a, n, f)
  if n <= 1 then
    f(a)
  else
    for i = 1, n do
      a[n], a[i] = a[i], a[n]
      permgen(a, n - 1, f)
      a[n], a[i] = a[i], a[n]
    end
  end
end
```

Coroutines and Iterators

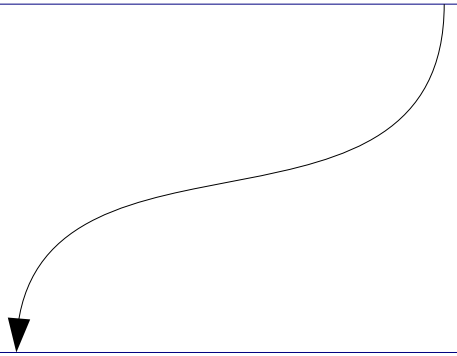
```
function permutations (a)
  return coroutine.wrap(function ()
           permgen(a, #a, coroutine.yield)
         end)
end

for a in permutations({1,2,3,4}) do
  printPerm(a)
end
```

Who is the Main Program

How to turn a complex interactive application into a library?

Who is the Main Program

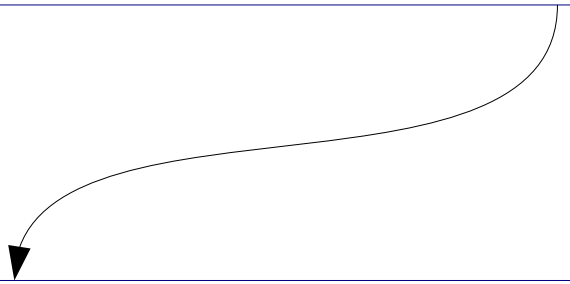
```
/* huge and complex application */
int main (int argc, char **argv) {
    ...
}
```



```
void readCommand (char *buff) {
  printf("enter command:\n");
  fgets(buff, MAX, stdin);
}
```

Who is the Main Program

```
/* huge and complex application */
int main (int argc, char **argv) {
  /* create coroutine with Lua script */
    ...
}
```



```
void readCommand (char *buff) {
  lua_resume(...);
  /* pass result to buffer */
  ...
}
```

Who is the Main Program

```
-- Lua script
emitCommand = coroutine.yield


emitCommand("doCommand1")
   ...
emitCommand("doCommand2")
   ...
emitCommand("doCommand3")
```

Coroutines x continuations

- Most uses of continuations can be coded with coroutines
  - coroutines ☺
  - “who has the main loop” problem
    - Producer-consumer
    - extending x embedding
  - iterators x generators
    - the same-fringe problem
  - collaborative multithreading

Coroutines x continuations

- Multi-shot continuations are more expressive than coroutines
- Some techniques need code reorganization to be solved with coroutines or one-shot continuations
  - e.g., oracle functions