Coroutines

Some Motivation

We would like to write a program that:

- Continually reads text from one file
- Continually writes this text to another file

We can define two functions:

- One which *produces* text from an input file (**producer**)
- One which *consumes* this text and prints it to an output file (**consumer**)

How do these two functions communicate?

Coroutines

Thread-like, but not concurrent

Collaborative: Transfer control between each other

yield: Suspend execution until **resume** is called

resume: Resume execution until next **yield** or end

Yield and resume can exchange data with each other

Producers and Consumers in Lua

```
function producer ()
  return coroutine.create(function ()                  -- create a new coroutine
    while true do                                      -- can also loop on another condition, etc.
      local value = produce()                          -- produce the value (e.g., read from a file)
      coroutine.yield(value)                           -- suspend execution until resumed, passing the value
    end
  end)
end

function consumer(prod)
  while true do
    local status, value = coroutine.resume(prod)       -- resume execution of the producer, getting the value
    consume(value)                                     -- consume the value (e.g., write to a file)
  end
end
```

Iterators

An iterator is just a **producer**, and the loop body is the **consumer**

No need to worry about maintaining state between calls to the iterator

Can also *compose* iterators:

- Traverse two data structures in a synchronized way
- Encapsulate the state

Example: Merge two binary search trees

Merge Two BSTs: Example

```
function inorder(node)                                  -- produce values via inorder traversal of a tree
  if (node)
    inorder(node.left)
    coroutine.yield(node.key)                           -- suspend until next iteration, passing the node key
    inorder(node.right)
  end
end

function iterator(tree)
  local prod = coroutine.create(function(t)             -- the producer coroutine
    inorder(t)
  end)
  return function()
    local status, key = coroutine.resume(prod, tree)    -- resume execution of the producer, getting the value
    return key                                          -- return the key to be consumed by the caller
  end
end
```

How might we build an iterator to merge two BSTs from this?

Merge Two BSTs: Producing Values

```
function merge(t1, t2)                                  -- produce values by merging two trees
   local it1 = iterator(t1)
   local it2 = iterator(t2)
   local v1 = it1()
   local v2 = it2()
   while (v1 || v2) do
      if (v1 != nil and (v2 == nil or v1 < v2)) then
         coroutine.yield(v1)                            -- suspend until next iteration, passing key from t1
         v1 = it1()                                     -- iterate t1
      else
         coroutine.yield(v2)                            -- suspend until next iteration, passing key from t2
         v2 = it2()                                     -- iterate t2
      end
 end

function mergeTreeIterator(t1, t2)
   -- …                                                 -- what might this look like?
end
```