Non-standard control flow and the stack

- Exceptions
- Coroutines and Threads
Exceptions

- Programs often need to handle exceptional conditions, i.e., deviations from “normal” control flow.

- Exceptions may arise from:
  - Failures of built-in or system operations (e.g., division by zero, reading past end of file).
  - User-defined events (e.g., key not found in dictionary).

- Handling these conditions “in-line” distorts the code for the “normal” case.

- Most recent languages (Ada, C++, Java, Python, etc.) provide a way to define, raise (or throw), and handle exceptions.
Scala exceptions example

class Help extends Exception  // define a new exception

try {
    ...
    if (gone wrong)
        throw new Help  // raise user-defined exception
    ...
    x = a / b  // might raise a built-in exception
    ...
} catch {
    case _: Help => ...report problem...
    case _: ArithmeticException => x = -99  // repair damage
}
Semantics of exceptions

- If there is a *statically* enclosing handler, throwing an exception behaves much like a `goto`.

  ```
  ... 
  if (gone wrong) goto help_label; 
  ...
  help_label: ...report problem...
  ```

- But what if no handler encloses the throw point?

  - In most languages, uncaught exceptions *propagate* to the next *dynamically* enclosing handler.
    - Caller can handle uncaught exceptions thrown in callee.
  
  - A few languages support *resumption* of the program at the throw point.
  
  - Many languages permit a *value* to be returned with the exception.
Dynamic exception handling

case class BadThing(problem: String) extends Exception

def foo() = {
  ... throw BadThing("my problem") ... 
}

def bar() {
  try {
    foo()
  } catch {
    case BadThing(problem) => println("oops:" + problem)
  }
}
Exceptions vs. Error Values

An alternative to user-thrown exceptions is to return status values, which must be checked by caller.

```scala
def find (k0:String,env:List[(String,Int)]) : Option[Int] = 
  env match {
    case Nil => None
    case (k,v)::t => if (k == k0)
      Some(v)
      else find(k0,t)
  }

  ...

  find("abc",e) match {
    case Some(v) => ... v ...
    case None => ...perform error recovery...
  }

Scala
Exceptions vs Error Values (2)

With exceptions, we can defer checking for (rare) error conditions until a more convenient point.

class NotFound extends Exception

def find (k0: String, env: List[(String, Int)]) : Int =
    env match {
        case Nil => throw new NotFound
        case (k, v)::t => if (k == k0)
            v
            else find(k0, t)
    }

... try {
    val v = find ("abc", e)
    ... v ...
} catch {
    case _: NotFound => ...perform error recovery...
}
Implementing exceptions (1)

- One approach to implementing exceptions is for runtime system to maintain a handler stack, with an entry for each currently active exception catching context.

- Each entry contains a handler code address and call stack pointer.

- When the scope of a handler is entered (e.g. by evaluating a `try...catch` expression), handler’s address is paired with current stack pointer and pushed onto handler stack.

- When an exception occurs, top of handler stack is popped, resetting the call stack pointer and passing control to the handler’s code. If this handler itself raises an exception, control passes to the next handler on the stack etc.

- Selective handlers work by simply re-throwing any exception they don’t want to handle (passing control to next handler on the stack).
Exceptions on purpose

In this model, throwing an exception provides a way to return quickly from a deep recursion, with no need to pop stack frames one at a time.

```ocaml
exception Zero
let rec product (l : int list) : int =
  let rec prod (l : int list) : int =
    match l with
    [] -> 1
    | 0::_ -> raise Zero
    | h::t -> h * prod t in
  try
    prod l
  with Zero -> 0
```

OCaml
Implementing Exceptions (2)

- The handler-stack implementation makes handling very cheap, but incurs cost each time we enter a new handler scope. If throws are very rare, this is a bad tradeoff.

- An alternative: runtime system uses a static table that maps each code address to the address of the statically enclosing handler (if any)

- If an exception occurs, table is inspected to find the appropriate handler

- If there is no handler covering the current address, runtime system looks for a handler that covers the return address (in the caller), and so on up the call stack
Coroutines

Coroutines are like subroutines that can transfer control back and forth without obeying the "last started, first finished" discipline.

They are very useful for maintaining multiple collaborating functions each with its own "local state"

- Convenient way to write iterators
- Generalize to cooperative threads
- ... and then to preemptive or genuinely concurrent threads
Example: coroutines in Lua

- co = coroutine.create(f)
  - create a new coroutine co that will run function f
- coroutine.resume(co)
  - pause and transfer control to coroutine co
- coroutine.yield()
  - pause and return control to coroutine that resumed us
- Can also pass values via resume and yield
Coroutine execution

function f()
  for i = 1,5 do
    print("in f",i)
    coroutine.yield()
  end
end

function g()
  co = coroutine.create(f)
  for i = 1,5 do
    print ("in g",i)
    coroutine.resume(co)
  end
end

executing g() produces

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in g1</td>
<td>in f1</td>
<td>in g2</td>
</tr>
<tr>
<td>in g2</td>
<td>in f2</td>
<td>in g3</td>
</tr>
<tr>
<td>in g3</td>
<td>in f3</td>
<td>in g4</td>
</tr>
<tr>
<td>in g4</td>
<td>in f4</td>
<td>in g5</td>
</tr>
<tr>
<td>in g5</td>
<td>in f5</td>
<td></td>
</tr>
</tbody>
</table>
Suppose we want to print the values in a binary tree in order:

```lua
function ptree(t)
if t ~= nil then
    ptree(t.l)
    print(t.v)
    ptree(t.r)
end
end
```

Lua

```
t =
{l={l=nil, v=1, r=nil}, v=2, r={l=nil, v=3, r={l=nil, v=4, r=nil}}}
```

`ptree(t)` prints `1 2 3 4`

What about performing an arbitrary action on the values in order?
Iterators from coroutines(2)

Abstracting out the tree walk using coroutines

```literate
function walk (t)
  if t ~= nil then
    walk(t.l)
    coroutine.yield(t.v)
    walk(t.r)
  end
end

function ptree (t)
  function cofun () walk (t) end
  local co = coroutine.create (cofun)
  while true do
    local _, res = coroutine.resume(co)
    if res == nil then break end
    print (res)
  end
end

ptree(t) prints 1 2 3 4
```
Iterators from coroutines(3)

Abstracting out the tree walk into an iterator

```lua
function walk (t)
    if t ~= nil then
        walk(t.l)
        coroutine.yield(t.v)
        walk(t.r)
    end
end

function treevals (t)
    function cofun () walk (t) end
    local co = coroutine.create(cofun)
    function iterator ()
        local _, res = coroutine.resume(co)
        return res
    end
    return iterator
end

function ptree(t)
    for i in treevals(t) do print(i) end
end

ptree(t) prints 1 2 3 4
```

first-class function!
mORE ON THIS LATER
Coroutines and Stacks

- To make yield and resume work, the language implementation must preserve the control state of each coroutine while the other is executing.

  - In particular, the state of the stack, with local variables, pending functions to return to, etc.

- This implies that we need one stack per coroutine.

- Switching between coroutines means swapping the current stack (typically a cheap operation at hardware level).

- Managing stack allocation and potential overflow is harder than in the usual single stack case.
From coroutines to threads

- Coroutines normally transfer control to each other explicitly.
- A common alternative is to support a collection of threads whose execution is controlled by a scheduler.
- Each thread periodically pauses its work and yields control back to the scheduler, which picks a paused thread to run next.
- These are called cooperative (non-preemptive) threads.
- If the threads share common data, they can take care to yield only when that memory is in a clean state.
Preemptive/concurrent threads

Many languages support genuinely concurrent threads that run simultaneously on multiple processors.

If these threads share data, they must be very careful to synchronize their accesses, e.g. using locks, to avoid accidental cross-thread corruption.

Preemptive multithreading gives the illusion of concurrency by sharing a single processor among multiple threads, using timer interrupts to force periodic yields.

Since threads no longer control when they yield, they must be just as careful about synchronization as in the genuinely concurrent case.