Iteration into Recursion

Any iteration can be written as a recursion, e.g.

```scala
while (c) {e}
```

is equivalent to

```scala
def f(b:Boolean):Unit =
  if (b) {
    e;
    f(c)
  }
  f(c)
```

assuming the variables used by `c` and `e` are in scope.
Recursion into iteration?

- When can we do the converse?

- A recursion can be rewritten as an iteration (without needing any extra storage) whenever all the recursive calls are in tail position.
  
  - Call in tail position iff it is the last thing performed by the caller before it itself returns.

- Often worthwhile, in order to avoid pushing a stack activation frame for each recursive call (lowers total stack needed and eliminates push/pop time).

- A decent compiler can turn tail-calls into iterations automatically. This is essential for functional languages, which use recursion heavily, but is useful even for imperative ones.
Scala list tail-call examples

```scala
def find (y:Int,xs:List[Int]):Boolean = xs match {
  case Nil => false
  case (x::xs1) => (x == y) || find(y,xs1) // tail-recursive
}

def length (xs:List[Int]):Int = xs match {
  case Nil => 0
  case (_::xs1) => 1 + length(xs1) // not tail-recursive
}

def length_tr (xs:List[Int]):Int = {
  // use an auxiliary function with an accumulating parameter
  def f (xs:List[Int],len:Int):Int = xs match {
    case Nil => len
    case (_::xs1) => f (xs1,len+1) // tail-recursive
  }
  f(xs,0)
}
```

A decent compiler can turn tail-calls into iterations, thus saving the cost of pushing an activation frame on the stack. This is essential for functional languages, and useful even for imperative ones.
Systematic Removal of Recursion

But what about general (non-tail) recursion?

One way to get a better appreciation for how recursion is implemented is to see what is required to get rid of it.

Additional explicitly-allocated memory space is needed!
typedef struct tree *Tree;
struct tree {
    int value;
    Tree left, right;
};

void printtree(Tree t) {
    if (t) {
        print(t->value);
        printtree(t->left);
        printtree(t->right);
    }
}
Remove **tail-recursion**.

```c
void printtree(Tree t) {
    top:
    if (t) {
        print(t->value);
        printtree(t->left);
        t = t->right;
        goto top;
    }
}
```
**Step 2**

Use explicit stack to replace non-tail recursion. Simulate behavior of compiler by pushing local variables and return address onto the stack **before** call and popping them back off the stack **after** call.

Assume this stack interface, specialized to use `Tree` as the stack element type.

```c
Stack empty;
void push(Stack s, Tree t);
Tree pop(Stack s);
bool isEmpty(Stack s);
```
Here there is only one local variable \((t)\) and the return address is always the same, so there’s no need to save it.

```c
void printtree(Tree t) {
    Stack s = empty;
    top:
    if (t) {
        print(t->value);
        push(s,t);
        t = t->left;
        goto top;
    }
    if (!isEmpty(s)) {
        t = pop(s);
        goto retaddr;
    }
}
```

Simplify by:

- Rearranging to avoid the `retaddr` label.
- Pushing right child instead of parent on stack.
- Replacing first `goto` with a `while` loop.

```c
void printtree(Tree t) {
    Stack s = empty;
    top:
        while (t) {
            print(t->value);
            push(s, t->right);
            t = t->left;
        }
        if (!(isEmpty(s))) {
            t = pop(s);
            goto top;
        }
}
```
Rearrange some more to replace outer goto with another while loop.
(This is slightly wasteful, since an extra push, stackempty check and pop are performed on root node.)

```c
void printtree(Tree t) {
    Stack s = empty;
    push(s,t);
    while(!(isEmpty(s))) {
        t = pop(s);
        while (t) {
            print(t->value);
            push(s,t->right);
            t = t->left;
        }
    }
}
```
A more symmetric version can be obtained by pushing and popping the left children too.

Compare this to the original recursive program.

```c
void printtree(Tree t) {
    Stack s = empty;
    push(s, t);
    while(!isEmpty(s)) {
        t = pop(s);
        if (t) {
            print(t->value);
            push(s, t->right);
            push(s, t->left);
        }
    }
}
```
We can also test for empty subtrees **before** we push them on the stack rather than after popping them.

```c
void printtree(Tree t) {
    Stack s = empty;
    if (t) {
        push(s, t);
        while(!(isEmpty(s))) {
            t = pop(s);
            print(t->value);
            if (t->right)
                push(s, t->right);
            if (t->left)
                push(s, t->left);
        }
    }
}
```
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 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.

```
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 ...
}
Exceptions vs Error Values (2)

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

```scala
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 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...with` 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.

Selecting 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.

```scala
class Zero extends Exception

def product(l: List[Int]): Int = {
  def prod(l: List[Int]): Int = l match {
    case Nil => 1
    case h::t => if (h==0) throw new Zero else h * prod(t)
  }
  try {
    prod(l)
  } catch {
    case _ : Zero => 0
  }
}
```
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