Iteration into Recursion

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

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

is equivalent to

```
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.

- This rewrite is 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
}
```

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

```scala
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);
    }
}

(Adapted from R. Sedgewick, *Algorithms*, 2nd ed.)
Remove **tail-recursion**.

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

```
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;
        }
    retaddr:
        t = t->right;
        goto top;
}
```
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);
        }
    }
}
```