

CS558 Programming Languages

Winter 2008

Lecture 8

ITERATION VS. RECURSION

Any iteration can be written as a recursion.

For example:

```
while (t) do e
```

is equivalent to

```
void f (bool b) {  
    if (b) then {  
        e;  
        f(t)  
    }  
}  
f (t)
```

where we assume that the variables used by *e* and *t* are global.

When can we do the converse? It turns out that a recursion can be rewritten as an iteration whenever all the recursive calls are in **tail position**. To be in tail position, the call must be the **last** thing performed by the caller before it itself returns.

TAIL-CALL EXAMPLES

List operations can often be made tail-recursive in this way:

```
(* tail-recursive *)  
fun last [x] = x  
  | last (x::xs) = last xs
```

```
(* not tail-recursive *)  
fun length [] = 0  
  | length (x::xs) = 1 + (length xs)
```

```
(* use accumulating parameter; now is tail-recursive *)  
fun length l =  
  let fun f ([],len) = len  
        | f (x::xs,len) = f (xs,len+1)  
      in f (l,0)  
      end
```

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 languages (like ML) that lack iteration, and useful even for those that have it (like C).

SYSTEMATIC REMOVAL OF RECURSION

(Adapted from Sedgewick, *Algorithms*, 2nd ed.. Examples in C.)

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.

Original program:

```
typedef struct tree *Tree;
struct tree {
    int value;
    Tree left, right;
};

void printtree(Tree t) {
    if (t) {
        printf("%d\n", t->value);
        printtree(t->left);
        printtree(t->right);
    }
}
```

STEP 1

Remove **tail-recursion**.

```
void printtree(Tree t) {
top:
    if (t) {
        printf("%d\n", 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:

```
Stack empty;  
void push(Stack s,void* t);  
(void*) pop(Stack s);  
int isEmpty(Stack s);
```

STEP 2 (CONT.)

Here there is only one local variable (t) and the return address is always the same, so there's no need to save it.

```
void printtree(Tree t) {
    Stack s = empty;
top:
    if (t) {
        printf("%d\n", t->value);
        push(s, t);
        t = t->left;
        goto top;
retaddr:
        t = t->right;
        goto top;
    }
    if (!(isEmpty(s))) {
        t = pop(s);
        goto retaddr;
    }
}
```

STEP 3

Simplify by:

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

```
void printtree(Tree t) {
    Stack s = empty;
top:
    while (t) {
        printf("%d\n", t->value);
        push(s, t->right);
        t = t->left;
    }
    if (!(isEmpty(s))) {
        t = pop(s);
        goto top;
    }
}
```

STEP 4

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.)

```
void printtree(Tree t) {
    Stack s = empty;
    push(s,t);
    while(!isEmpty(s)) {
        t = pop(s);
        while (t) {
            printf("%d\n",t->value);
            push(s,t->right);
            t = t->left;
        }
    }
}
```

STEP 5

A more symmetric version can be obtained by pushing and popping the left children too.

Compare this to the original recursive program.

```
void printtree(Tree t) {
    Stack s = empty;
    push(s,t);
    while(!(isEmpty(s))) {
        t = pop(s);
        if (t) {
            printf("%d\n",t->value);
            push(s,t->right);
            push(s,t->left);
        }
    }
}
```

STEP 6

We can also test for empty subtrees **before** we push them on the stack rather than after popping them.

```
void printtree(Tree t) {
    Stack s = empty;
    if (t) {
        push(s,t);
        while(!isEmpty(s)) {
            t = pop(s);
            printf("%d\n",t->value);
            if (t->right)
                push(s,t->right);
            if (t->left)
                push(s,t->left);
        }
    }
}
```