PROCESSES AND FUNCTIONS

- Subroutines: avoid duplicating frequently used code
- Procedural abstraction: divide programs into named components with hidden internals
- Provide framework for managing local data, especially in recursive programs

Invoking a procedure generates a procedure activation, which has associated data such as:
- the return address of the caller
- the actual values corresponding to the formal parameters of the procedure
- space for the values of local variables associated with the procedure.

The data associated with each activation is independent from that for all other activations, so many activations can exist at once, as happens in a recursive program.

ACTIVATION FRAMES

In most languages, activation data can be stored on a stack, and we speak of pushing and popping activation frames from the stack, which is a very efficient way of managing local data.

A typical activation stack, shown just before inner call to f returns.

WHAT ABOUT REGISTERS?

Although it is convenient to view all locations as memory addresses, most machines also have registers, which are:
- much faster to access,
- but very limited in number (e.g., 4 to 64).

So compilers try to keep variables (and pass parameters) in registers when possible, but always need memory as a backup. Using registers is fundamentally just an (important!) optimization.

Easy to have environment map each name to location that is either memory address or register.
- But registers don’t have addresses, so they can’t be accessed indirectly, and register locations can’t be passed around or stored.
TAIL-CALL OPTIMIZATION

Any iteration can be written as a recursion.

For example:

while \( t \) do \( e \)

is equivalent to

\[
\text{void } f \text{ (bool } b) \text{ \{} \\
\text{ if (} b \text{) then \{} \\
\text{ \quad } e; \\
\text{ \quad } f(t) \\
\text{ \}\} \\
\text{ }} \\
\text{f (} t \text{)}
\]

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:

\[
\begin{align*}
\text{(* tail-recursive *)} \\
\text{fun last [} x \text{] = } x \\
\text{ \quad last (} x::xs \text{) = last xs} \\
\text{(* not tail-recursive *)} \\
\text{fun length [} [] \text{] = 0} \\
\text{ \quad length (} x::xs \text{) = 1 + (length xs)} \\
\text{(* use accumulating parameter; now is tail-recursive *)} \\
\text{fun length 1 = } \\
\text{ \quad let fun f ([],len) = len} \\
\text{ \quad \quad | f (} x::xs,\text{len) = f (} xs,\text{len+1)} \\
\text{ \quad \quad in f (1,0)} \\
\text{ end}
\end{align*}
\]

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.)

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:

\[
\begin{align*}
\text{typedef struct tree *Tree;} \\
\text{struct tree \{} \\
\text{ \quad int value;} \\
\text{ \quad Tree left;} \\
\text{ \quad Tree right;} \\
\text{ \};} \\
\text{\quad void printtree(Tree t) \{} \\
\text{ \quad if (t) \{} \\
\text{ \quad \quad printf(} "%d\n",t->value); \\
\text{ \quad printtree(t-left); \\
\text{ \quad printtree(t-right); \\
\text{ \quad \}\} \\
\text{}} \\
\end{align*}
\]

Remove tail-recursion.

\[
\begin{align*}
\text{void printtree(Tree t) \{} \\
\text{ top:} \\
\text{ \quad if (t) \{} \\
\text{ \quad \quad printf(} "%d\n",t->value); \\
\text{ \quad print(t-left); \\
\text{ \quad t = t->right; \\
\text{ \quad goto top; \\
\text{ \quad \}\} \\
\text{}} \\
\end{align*}
\]
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:

```c
Stack empty;
void push(Stack s, void* t);
(void*) pop(Stack s);
int 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) {
        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.

```c
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.)

```c
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;
        }
    }
}
```
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) {
            printf("%d\n",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);
            printf("%d\n",t->value);
            if (t->right)
                push(s,t->right);
            if (t->left)
                push(s,t->left);
        }
    }
}
```