Each invocation of a procedure requires 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.

This activation data must live from the time the procedure is invoked until the time it returns. If one procedure calls another procedure, their activation data must be kept separate, because their lifetimes overlap. In particular, the data for all invocations of a recursive procedure must be kept separate.
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.

Program:

```plaintext
int f(int x, int y){
    int z = y+y;
    if (z > 0)
        z = f(z,0);
    return z+y;
}

void main() {
    int w = 10;
    w = f(w,w);
}
```
**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 a (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.
Any iteration can be written as a recursion.

For example:

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

is equivalent to

```c
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 to use just jumps 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.
IDENTIFYING TAIL CALLS

Tail-recursive:

```scheme
(define (last xs)
  (cond
    [(empty? xs) (error 'last "empty list")]
    [(empty? (rest xs)) (first xs)]
    [else (last (rest xs))])))
```

Not tail-recursive:

```scheme
(define (length xs)
  (if (empty? xs)
      0
      (+ 1 (length (rest xs)))))
```

Note that the recursive call in `length` is **not** a tail call (even though it syntactically appears at the end of the function text) because `length` does more work (adding 1) after the recursive call returns.
Here’s an alternative implementation of \texttt{length} that is tail-recursive:

\begin{verbatim}
(define (alength xs len)
  (if (empty? xs)
      len
      (alength (rest xs) (+ 1 len))))

(define (length:2 xs) (alength xs 0))
\end{verbatim}

List operations can often be made tail-recursive by adding an accumulating parameter in this way.
A decent compiler can turn tail-calls into ordinary jumps, thus saving the cost of pushing an activation frame on the stack. For example, the computation of alength should not grow the stack at all. This is essential for languages (like Scheme) that rely heavily on recursion, and useful even for those that have it (like 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:

```c
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);
    }
}
```
Remove **tail-recursion**.

```c
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:

```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;
    }
    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) {
            printf("%d\n", 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) {
        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);
        }
    }
}
```
It turns out that if our language has higher-order functions, we can code in a style where every call is a tail call!

To show the approach, let’s go back to our familiar non-tail-recursive list length function. We wrap it in a function that prints out the result:

```scheme
(define (length xs)
  (if (empty? xs)
      0
      (+ 1 (length (rest xs)))))

(define (plength xs)
  (printf "the length of \~a is \~a" xs (length xs)))
```
Here’s another tail-recursive way to write this:

```scheme
(define (klength xs k)
  (if (empty? xs)
      (k 0)
      (klength (rest xs) (lambda (r) (k (+ 1 r))))))

(define (plength:2 xs)
  (klength xs (lambda (r)
      (printf "the length of ~a is ~a" xs r))))
```

This rather odd code was constructed by giving `klength` an additional argument, `k`, of type `int → void`. Instead of returning its “result” value, `klength` passes it downwards to `k`.

Notice that `every` call in `klength` is a **tail-call**. Note too that `klength` only returns after `printf` is invoked; in essence, it needn’t return at all. This is because `k` is serving the same role as a return address: saying “what to do next.”

This means we can evaluate `klength` **without a stack**!
COMPARISON OF COMPUTATIONS

(print (length '(a b))) →
(print (+ 1 (length '(b)))) →
(print (+ 1 (+ 1 (length '())))) →
(print (+ 1 (+ 1 0))) →
(print (+ 1 1)) →
(print 2)

(klength '1 2) (lambda (r) (print r))) →
(klength '2 (lambda (r_1) ((lambda (r) (print r)) (+ 1 r_1)))) →
(klength () (lambda (r_2))
    ((lambda (r_1) ((lambda (r) (print r)) (+ 1 r_1)) (+ 1 r_2)))) →
    ((lambda (r_2) ((lambda (r_1) ((lambda (r) (print r)) (+ 1 r_1)) (+ 1 r_2))) 0) →
(klength (r_1) ((lambda (r) (print r)) (+ 1 r_1))) 0) →
(klength (r_1) ((lambda (r) (print r)) (+ 1 r_1))) 1) →
(klength (r) (print r)) (+ 1 1) →
(klength (r) (print r)) 2) →
(print 2)
Functions like \( k \) are (loosely) called continuations and programs written using them are said to be in (a form of) continuation-passing style (CPS).

We may choose to write (parts of) programs explicitly in CPS because it makes it easy to express a particular algorithm or because it clarifies the control structure of the program (more later).

Note that CPS’ed Scheme programs are just a subset of ordinary Scheme programs that happens to make heavy use of the (existing) enormous power of first-class functions.

Remarkably, we can also systematically convert any functional program into an equivalent CPS program. See IALC Chapter 18.

But first, why “continuation”?
Continuations

Broadly speaking, a continuation is a representation of a dynamic program point, i.e., the state of the program at some particular time during execution.

A program’s state at a particular dynamic point must contain exactly the information needed to “continue” execution of the program from that point until the program completes.

If we are describing execution in terms of a low-level machine, we can think of a continuation for a program point as the machine state at that point: the values of the pc, the sp, and other registers.

If we are describing execution in a an interpreter implemented in a functional language, we can think of a continuation for a program point as a function, which, given the value of the expression being computed at that point, returns the result of the entire program.
Implicit and Explicit Continuations

Our definition of continuations refers to a description of internal state. These continuations are implicit in our execution model of the program. For example, the functions just referred to exist in the interpreter for a program, not in the program itself.

The function $k$ we introduced in `klength` is not a continuation in this sense; it appears explicitly in the program. But we can justify calling it a continuation because it does correspond to an (implicit) continuation in the original program `length`!

In fact, we can view the process of rewriting `length` into `klength` as one of making internal continuations explicit. Systematic rewriting of this kind may be a useful step during language compilation, because programs in CPS may be easier to process than ordinary “direct-style” programs—for example, they can be executed without a runtime stack. Some compilers for Scheme and other functional languages do this.
Some languages (including Scheme) permit us to “capture” the current dynamic program state (implicit continuation) into an explicit data object. This is done using a special expression form such as \texttt{call/cc} (for “call-with-current-continuation”) or \texttt{let/cc}, which binds a representation of the current state to a variable.

These so-called \textbf{first-class continuations} behave much like functions (indeed in Scheme they \texttt{are} functions): invoking one restores the program’s state to the time of capture. This is very handy for implementing user-defined control operators such as exceptions and coroutines.

Note that C has a similar mechanism in the form of \texttt{setjmp/longjmp}; the \texttt{jmp_buf} effectively contains a concrete representation for a continuation!

To the programmer, a first-class continuation is a black box (just like an ordinary function); it can be passed around and invoked, but its internals cannot be inspected. The actual representation, and the cost of performing captures and resets of the current state, depend on the compiler’s implementation technology.
CONTINUATIONS AT MANY LEVELS

In summary, the term “continuation” can be used in many different, though related, ways, which we should try to keep straight:

- We can program in an explicit “continuation-passing” style, as exemplified by \texttt{klength}.
- We can convert ordinary (“direct style”) programs into CPS representations in order to make them more “machine-like” and/or easier to optimize.
- If our language permits, we can “capture” or “reify” program state (the current implicit continuation) into an explicit data object (a “first-class continuation”) which can later be used to restore that state.
- Abstract continuation functions may be used to describe the denotation semantics of a language (but we won’t explore this).