Let’s say you’ve written a little programming language and an interpreter for it. It begins to get some users and you expand the language. You add first-class functions. You then add a type system. Now you have a few more users and they start to complain that it’s too slow. Time to write a compiler.

Your first instinct might be to try to write one from scratch. One problem with this method is that the correctness of your interpreter (which has been developed over time and stress-tested by your users) is not necessarily preserved in your compiler.

It would be nice if we had a way to automatically transform an interpreter into a compiler. In fact, there is a way—the CPS transformation.

In these notes and several that follow, we will “hand compile” several programs. We won’t transform all the way down to assembly code, but you’ll get the spirit of how that could be done.

Let’s begin the way so many computer scientists do—with the factorial program:

### Compiling Factorial

```
(define !
  (lambda (n)
    (if (zero? n)
      1
      (* n (! (- n 1))))))
```

We CPS this program in the standard way:

```
(define !/k
  (lambda (n k)
    (if (zero? n)
      (k 1)
      (!/k (- n 1)))
```
We can view the continuation parameter \( k \) in this function as a stack. Application of \( k \) is a pop and augmenting \( k \) is a push. Let’s make this stack explicit:

\[
(\lambda (v) (k (* n v))))
\]

We can recover the actual factorial function with the following code:

\[
(define (!n)
  (!/stack n EmptyStack))
\]

In our quest to move lower and lower in the abstraction hierarchy, let’s change our representation of stack to be a list of structures:

A stack record is the type of thing that the stack contains. We will have two different types of stack records for this problem, \texttt{sr/mult} and \texttt{sr/empty}. The latter is used simply to signal the end of the stack:

\[
(define-sr/mult (num))
\]

\[
(define-sr/empty ())
\]

\[
(define (Pop stack value)
  (let ([top-ar (first stack)])
    (cond
      [(sr/mult? top-ar)
        (Pop (rest stack)
          (lambda (v) (Pop stack (* n v)))))
      []
    )))
\]
As a side-point, note that we can transform the CPS version into the accumulator form of introductory programming classes. To do this, note that we need intelligence of what's on the stack, and we have to also assume associativity:

```scheme
(define !/stack/struct
  (lambda (n stack)
    (if (zero? n)
        (Pop stack 1)
        (!/stack/struct (− n 1)
          (Push stack n))))))
```

```scheme
(define Pop *)
(define Push *)
(define EmptyStack 1)
(define (! n)
  (!/stack/struct n EmptyStack))
```

**Compiling Tree-Sum**

The original function is as follows:

```scheme
(define-struct tree/mt ()
(define-struct tree/node (n left right))

(define (tree-sum t)
  (cond
    [(tree/mt? t) 0]
    [(tree/node? t)
      (+ (tree/node-n t)
          (tree-sum (tree/node-left t))
          (tree-sum (tree/node-right t)))]))
```

```scheme
(define (tree-sum t)
  (cond
    [(tree/mt? t) 0]
    [(tree/node? t)
      (+ (tree/node-n t)
          (tree-sum (tree/node-left t))
          (tree-sum (tree/node-right t)))]))
```

```scheme
(define (tree-sum t)
  (cond
    [(tree/mt? t) 0]
    [(tree/node? t)
      (+ (tree/node-n t)
          (tree-sum (tree/node-left t))
          (tree-sum (tree/node-right t)))]))
```
We now perform the standard CPS transformation:

```scheme
(define-struct tree/mt ())
(define-struct tree/node (n left right))

(define (tree-sum/k t k)
  (cond
   [(tree/mt? t) (k 0)]
   [(tree/node? t)
    (tree-sum/k (tree/node-left t)
      (lambda (v-left)
        (tree-sum/k (tree/node-right t)
          (lambda (v-right)
            (k (+ (tree/node-n t) v-left v-right))))))])

(define (tree-sum t)
  (tree-sum/k t (lambda (v) v)))
```

Just as in the last example, we will make the stack explicit and represent it as a list of structures:

```scheme
(define-struct tree/mt ())
(define-struct tree/node (n left right))

In this case, we have three different kinds of stack records:

```scheme
(define-struct sr/empty ())
(define-struct sr/left-add (t))
(define-struct sr/right-add (t left-value))
```

Why do we need a different stack record type for the left and the right side of the tree?

```scheme
(define (tree-sum/stack t stack)
  (cond
   [(tree/mt? t) (Pop stack 0)]
   [(tree/node? t)
    (tree-sum/stack (tree/node-left t)
      (Push stack
        (make-sr/left-add t)))]

(define (Pop stack value)
  (let ([top-ar (first stack)])
    (cond
     [(sr/left-add? top-ar) top-ar]
     [(sr/right-add? top-ar)
      (Pop (rest stack))])))
```

4
( + (tree/node-n (sr/right-add-t top-ar))
   (sr/right-add-left-value top-ar)
   value))]
[(sr/empty? top-ar)
 value]))))

(define (Push stack record)
  (cons record stack))

(define EmptyStack
  (cons (make-sr/empty) empty))

(define (tree-sum t)
  (tree-sum/stack t EmptyStack))