Recursion

Ken Q Pu, Faculty of Science, Ontario Tech University

Basic form of recursion

We can define a function with a *local name* that can be used inside the body to support recursion.

(**fn** <*name*> [<*parameters...*>] <body>)

; A very inefficient implementation of factorial

```
(fn fac [n]
  (cond (zero? n) 1
        :else (* n (fac (- n 1)))))
```

This form of recursion is highly inefficient, and should almost never be used.

Excessive usage of memory during recursion.

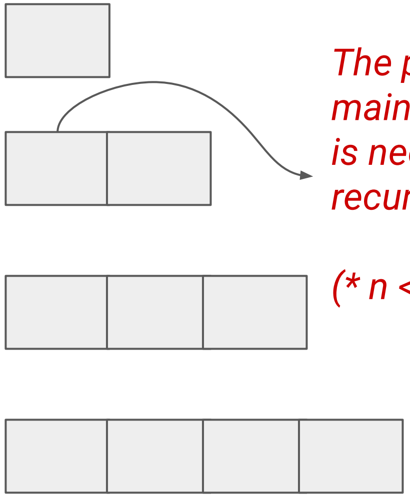
Why recursion is expensive?

Each function invocation requires some memory usage, known as the stack of the function call.

This is the design of modern operating systems and the Java virtual machine.

```
(def factorial
  (fn fac [n]
    (cond (zero? n) 1
          :else (* n (fac (- n 1))))))
```

```
(factorial 3)
(* 3 (factorial 2))
(* 2 (factorial 1))
(* 1 (factorial 0))
```



The previous stack is maintained because it is needed ***after*** *the recursion call.*

(n <recursion>)*

Ken Q Pu, Faculty of Science, Ontario Tech University

Recursion without bound

There is only finite memory for the stacks.

When recursion depth exceeds the maximum memory capacity, we hit the runtime memory error, famously known as:

Stack Overflow

Tail Recursion

When the function returns on a recursive call, then the recursion is called a tail recursion.

Namely, **no more computation is permitted after the recursive call.**

```
(def factorial
  (fn self [n]
    (cond (zero? n) 1
          :else (* n (self (- n 1))))))
```

Recursive call

Further computation performed after the recursive call.

So, this implementation of factorial is not a tail recursion.

Tail Recursion

```
(def factorial
 (fn self [count-down result]
  (cond (zero? count-down) result
        :else (self (dec count-down) (* result count-down)))))
```

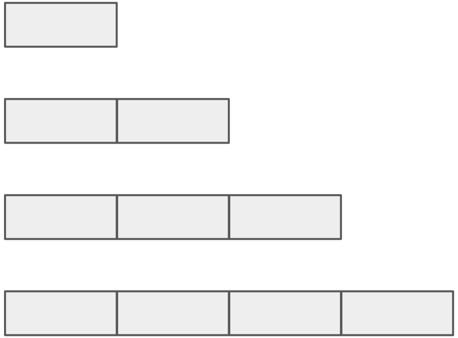
```
(factorial 3 1)

(factorial 2 (* 1 3))

(factorial 1 (* 1 3 2))

(factorial 0 (* 1 3 2 1))

(* 1 3 2 1)
```



Tail Recursion

```
(def factorial
 (fn self [count-down result]
  (cond (zero? count-down) result
        :else (self (dec count-down) (* result count-down)))))
```

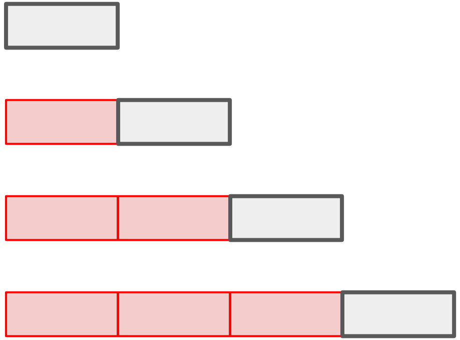
```
(factorial 3 1)

(factorial 2 (* 1 3))

(factorial 1 (* 1 3 2))

(factorial 0 (* 1 3 2 1))

(* 1 3 2 1)
```



Ken Q Pu, Faculty of Science, Ontario Tech University

Tail Recursion

```clojure
(def factorial
 (fn self [count-down result]
  (cond (zero? count-down) result
        :else (self (dec count-down) (* result count-down)))))
```

```clojure
(factorial 3 1)

(factorial 2 (* 1 3))

(factorial 1 (* 1 3 2))

(factorial 0 (* 1 3 2 1))

(* 1 3 2 1)
```

If we can inform Clojure that the recursion is a tail recursion, then Clojure can run recursion far more efficiently.

The **recur** form

The **recur**-form performs a recursion call in a tail recursion situation.

It can be used in both function body or a **loop**-form.

```
(recur <arguments...>)
```

The recur-form will start evaluating at the recursion point with the new values specified in <*arguments...*>

Using recur

```
(def factorial
 (fn self [count-down result]
  (cond (zero? count-down) result
        :else (self (dec count-down) (* result count-down)))))
```

```
(def factorial
 (fn [count-down result]
  (cond (zero? count-down) result
        :else (recur (dec count-down) (* result count-down)))))
```

The **loop**-form

The loop form allows us to perform recursion as expression rather than a function declaration.

It involves:

1. Initialize recursion symbols at the start of the recursion.
2. Define a loop-body that uses **recur** to start the next iteration with new values to the recursion symbols.
3. Make sure the loop-body eventually evaluates to data other than the **recur** form.

```
(loop [<x> <value>
       <y> <value>]
  (...
     (recur <value> <value>))
```

*Since **recur** can only be used for tail recursion, we cannot pass (recur ...) into any other function applications.*

Ken Q Pu, Faculty of Science, Ontario Tech University

Factorial done right

```
(def factorial
 (fn [n]
  (loop [i n
         accumulator 1]
    (zero? i) accumulator
    :else (recur (dec i) (* accumulator i)))))
```

Now we can use the factorial function naturally with:

`(factorial 100)`

We also don't need to worry about stack overflow errors for large inputs.