Higher-Order Procedures

In the functional paradigm, functions themselves can be processed as data.

In the functional paradigm, functions themselves can be processed as data.

In the same way you can pass data values as parameters in a function, you can pass function as a parameter in another function.

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions similar?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions similar?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions similar?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions similar?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions similar?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions similar?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions different?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions different?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

How are the functions different?

```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))

(define stack-copies-of
  (lambda (quantity image)
    (if (= quantity 1)
        image
        (stack (stack-copies-of (- quantity 1) image) image))))
```

The form is the same. The functions are different.

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                     (- quantity 1)
                                     thing)
                 thing))))
```

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                     (- quantity 1)
                                     thing)
                 thing))))
```

The first parameter in the function `together-copies-of` is a function.

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                     (- quantity 1)
                                     thing)
                 thing))))


(define stack-copies-of
  (lambda (quantity image)
    (together-copies-of stack quantity image)))
```

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                     (- quantity 1)
                                     thing)
                 thing))))
```

```
(define stack-copies-of
  (lambda (quantity image)
    (together-copies-of stack quantity image)))
```

Actual parameter

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                     (- quantity 1)
                                     thing)
                 thing))))


(define stack-copies-of
  (lambda (quantity image)
    (together-copies-of stack quantity image)))
```

Formal parameter

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                     (- quantity 1)
                                     thing)
                 thing))))
```

```
(define stack-copies-of
  (lambda (quantity image)
    (together-copies-of stack quantity image)))
```

What is the definition of `power`?

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                     (- quantity 1)
                                     thing)
                 thing))))


(define stack-copies-of
  (lambda (quantity image)
    (together-copies-of stack quantity image)))


(define power
  (lambda (base exponent)
    (together-copies-of * exponent base)))
```

```
(define num-digits-in-satisfying
  (lambda (n test?)
    (cond ((< n 0)
           (num-digits-in-satisfying (- n) test?))
          ((< n 10)
           (if (test? n) 1 0))
          ((test? (remainder n 10))
           (+ (num-digits-in-satisfying (quotient n 10) test?)
              1))
          (else
           (num-digits-in-satisfying (quotient n 10) test?)))))
```

What is the definition of `num-odd-digits`?

```
(define num-digits-in-satisfying
  (lambda (n test?)
    (cond ((< n 0)
           (num-digits-in-satisfying (- n) test?))
          ((< n 10)
           (if (test? n) 1 0))
          ((test? (remainder n 10))
           (+ (num-digits-in-satisfying (quotient n 10) test?)
              1))
          (else
           (num-digits-in-satisfying (quotient n 10) test?)))))

(define num-odd-digits
  (lambda (n)
    (num-digits-in-satisfying n odd?)))
```

```
(define num-digits-in-satisfying
  (lambda (n test?)
    (cond ((< n 0)
           (num-digits-in-satisfying (- n) test?))
          ((< n 10)
           (if (test? n) 1 0))
          ((test? (remainder n 10))
           (+ (num-digits-in-satisfying (quotient n 10) test?)
              1))
          (else
           (num-digits-in-satisfying (quotient n 10) test?)))))

(define num-odd-digits
  (lambda (n)
    (num-digits-in-satisfying n odd?)))
```

What is the definition of `num-6s`?

```
(define num-digits-in-satisfying
  (lambda (n test?)
    (cond ((< n 0)
           (num-digits-in-satisfying (- n) test?))
          ((< n 10)
           (if (test? n) 1 0))
          ((test? (remainder n 10))
           (+ (num-digits-in-satisfying (quotient n 10) test?)
              1))
          (else
           (num-digits-in-satisfying (quotient n 10) test?)))))

(define num-odd-digits
  (lambda (n)
    (num-digits-in-satisfying n odd?)))

(define num-6s
  (lambda (n)
    (num-digits-in-satisfying n (lambda (m) (= m 6)))))
```

The Halting Problem

The Halting Problem

Is it possible to write a program that does halt, that can determine whether any other program would halt if it were executed?

The Halting Problem

Is it possible to write a program that does halt, that can determine whether any other program would halt if it were executed?

NO!

The Halting Problem

```
(define return-seven
  (lambda ()
    7))
```

The Halting Problem

```
(define return-seven
  (lambda ()
    7))

(define loop-forever
  (lambda ()
    (loop-forever)))
```

The Halting Problem

```
(define return-seven
  (lambda ()
    7))

(define loop-forever
  (lambda ()
    (loop-forever)))

(define halts?
  (lambda (alpha)
    #t ; Bug. Should return #t if alpha halts, otherwise #f
    ))
```

The Halting Problem

```
(define return-seven
  (lambda ()
    7))

(define loop-forever
  (lambda ()
    (loop-forever)))

(define halts?
  (lambda (alpha)
    #t ; Bug. Should return #t if alpha halts, otherwise #f
    ))


> (halts? return-seven)
#t
> (halts? loop-forever)
#f
```

The Halting Problem

Proof by contradiction. Assume it is possible to write `halts?`, and show that assumption leads to a contradiction.

The Halting Problem

Proof by contradiction. Assume it is possible to write `halts?`, and show that assumption leads to a contradiction.

Construct function `debunk-halts?`

The Halting Problem

Proof by contradiction. Assume it is possible to write `halts?`, and show that assumption leads to a contradiction.

Construct function `debunk-halts?`

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(a) `debunk-halts?` halts

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(a) `debunk-halts?` halts

⇒ `(halts? debunk-halts?)` returns `#t`

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(a) `debunk-halts?` halts

⇒ `(halts? debunk-halts?)` returns `#t`

⇒ `loop-forever` executes

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(a) `debunk-halts?` halts

⇒ `(halts? debunk-halts?)` returns `#t`

⇒ `loop-forever` executes

⇒ `debunk-halts?` does not halt

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(a) `debunk-halts?` halts

$\Rightarrow$ `(halts? debunk-halts?)` returns `#t`

$\Rightarrow$ `loop-forever` executes

$\Rightarrow$ `debunk-halts?` does not halt

Contradiction

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(b) `debunk-halts?` does not halt

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(b) `debunk-halts?` does not halt

⇒ `(halts? debunk-halts?)` returns `#f`

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(b) `debunk-halts?` does not halt

⇒ `(halts? debunk-halts?)` returns `#f`

⇒ `return-seven` executes

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(b) `debunk-halts?` does not halt

⇒ `(halts? debunk-halts?)` returns `#f`

⇒ `return-seven` executes

⇒ `debunk-halts?` does halt

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

There are two possibilities:

(b) `debunk-halts?` does not halt

⇒ `(halts? debunk-halts?)` returns `#f`

⇒ `return-seven` executes

⇒ `debunk-halts?` does halt

Contradiction

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        (return-seven))))
```

Conclusion:

The assumption implies a contradiction in all possible scenarios.
Therefore, the assumption is false, and it is impossible to write `halts?`

Procedure factories

```
> (double 4)
8
> (double 5)
10
> (triple 4)
12
> (triple 5)
15
```

Procedure factories

```
> (double 4)
8
> (double 5)
10
> (triple 4)
12
> (triple 5)
15
```

```
(define double
  (make-multiplier 2))

(define triple
  (make-multiplier 3))
```

Procedure factories

```
(define double
  (make-multiplier 2))

(define triple
  (make-multiplier 3))
```

Define `make-multiplier`

Procedure factories

```
(define double
  (make-multiplier 2))

(define triple
  (make-multiplier 3))
```

Define `make-multiplier`

```
(define make-multiplier
```

Procedure factories

```
(define double
  (make-multiplier 2))

(define triple
  (make-multiplier 3))
```

Define `make-multiplier`

```
(define make-multiplier
  (lambda (scaling-factor)
```

Procedure factories

```
(define double
  (make-multiplier 2))

(define triple
  (make-multiplier 3))
```

Define `make-multiplier`

```
(define make-multiplier
  (lambda (scaling-factor)
    (lambda (x)
```

Procedure factories

```
(define double
  (make-multiplier 2))

(define triple
  (make-multiplier 3))
```

Define `make-multiplier`

```
(define make-multiplier
  (lambda (scaling-factor)
    (lambda (x)
      (* x scaling-factor))))
```

If the factory manufactures a function that calls itself the function must be named.

```
(define function-factory
  (lambda (parameter-for-factory)
    (define function-returned
      (lambda (parameter-for-function-returned)
           ...
             recursive call to function-returned
           ...                                      ))
    function-returned))
```

```
> (repeatedly-square 2 0)
2
> (repeatedly-square 2 1)
4
> (repeatedly-square 2 2)
16
> (repeatedly-square 2 3)
256
```

```
> (repeatedly-square 2 0)
2
> (repeatedly-square 2 1)
4
> (repeatedly-square 2 2)
16
> (repeatedly-square 2 3)
256
```

Function returned by the factory

```
(repeatedly-square 2 3)
```

```
> (repeatedly-square 2 0)
2
> (repeatedly-square 2 1)
4
> (repeatedly-square 2 2)
16
> (repeatedly-square 2 3)
256
```

Function returned by the factory

```
(repeatedly-square 2 3)
```

Two parameters

```
> (repeatedly-square 2 0)
2
> (repeatedly-square 2 1)
4
> (repeatedly-square 2 2)
16
> (repeatedly-square 2 3)
256
```

Function returned by the factory

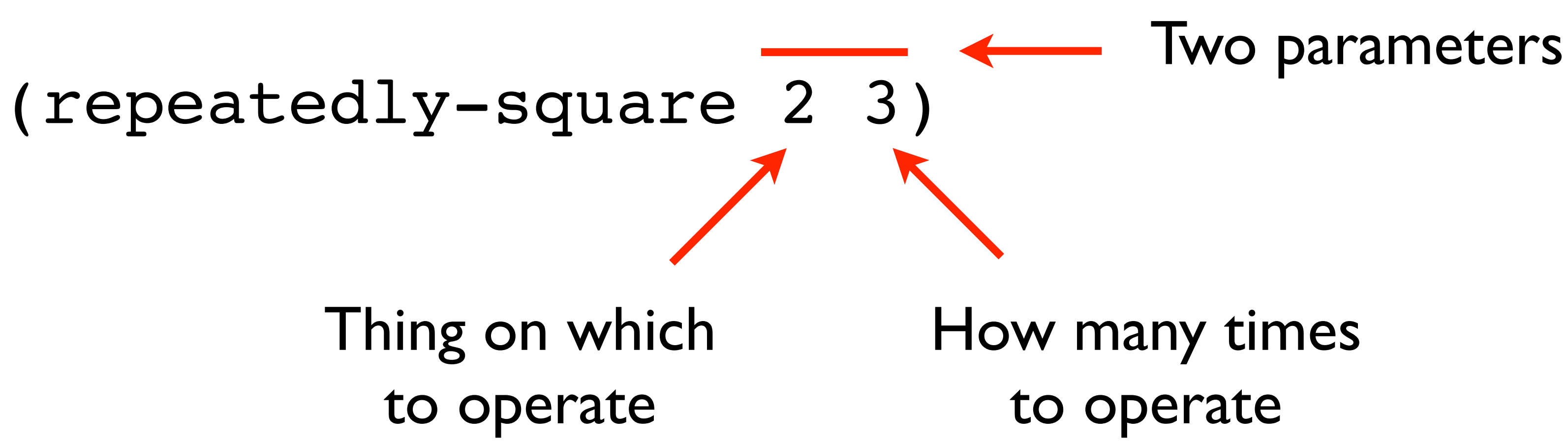
```
(repeatedly-square 2 3)
```

Two parameters

Thing on which to operate

```
> (repeatedly-square 2 0)
2
> (repeatedly-square 2 1)
4
> (repeatedly-square 2 2)
16
> (repeatedly-square 2 3)
256
```

Function returned by the factory



```
> (repeatedly-square 2 3)
256
```

Function returned by the factory

```
(repeatedly-square 2 3)
```

Calling the factory to make the function

```
> (repeatedly-square 2 3)
256
```

Function returned by the factory

```
(repeatedly-square 2 3)
```

Calling the factory to make the function

```
(define repeatedly-square
  (make-repeated-version-of sqr))
```

```
> (repeatedly-square 2 3)
256
```

Function returned by the factory

```
(repeatedly-square 2 3)
```

Calling the factory to make the function

```
(define repeatedly-square
  (make-repeated-version-of sqr))
```

Returns a function
having two parameters

```
> (repeatedly-square 2 3)
256
```

Function returned by the factory

```
(repeatedly-square 2 3)
```

Calling the factory to make the function

```
(define repeatedly-square
  (make-repeated-version-of sqr))
```

Returns a function having two parameters (make-repeated-version-of)

Has only one parameter itself, the operation (sqr)

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
  (lambda (f)  ; make a repeated version of f
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
  (lambda (f)  ; make a repeated version of f
    (define the-repeated-version
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
  (lambda (f)  ; make a repeated version of f
    (define the-repeated-version
      (lambda (b n)  ; which does f n times to b
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
  (lambda (f)  ; make a repeated version of f
    (define the-repeated-version
      (lambda (b n)  ; which does f n times to b
        (if (= n 0)
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
  (lambda (f)  ; make a repeated version of f
    (define the-repeated-version
      (lambda (b n)  ; which does f n times to b
        (if (= n 0)
            b
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
  (lambda (f)  ; make a repeated version of f
    (define the-repeated-version
      (lambda (b n)  ; which does f n times to b
        (if (= n 0)
            b
            (the-repeated-version (f b) (- n 1)))))
```

```
(repeatedly-square 2 3)

(define repeatedly-square
  (make-repeated-version-of sqr))
```

Define the factory

```
(define make-repeated-version-of
  (lambda (f)  ; make a repeated version of f
    (define the-repeated-version
      (lambda (b n)  ; which does f n times to b
        (if (= n 0)
            b
            (the-repeated-version (f b) (- n 1)))))
    the-repeated-version))
```