

Higher-Order Procedures

(in Ruby)

based on 'Structure and Interpretation of Computer Programs' (1985 MIT Press)
by Hal Abelson and Gerald Jay Sussman.
<http://swiss.csail.mit.edu/classes/6.001/abelson-sussman-lectures/>

<http://www.natemurray.com>

Nathan Murray <nate@natemurray.com> v1.0 12/13/06

legal

The copy in this presentation is taken directly from Structure and Interpretation of Computer Programs by Hal Abelson and Gerald Jay Sussman (MIT Press, 1984; ISBN 0-262-01077-1). Specifically section 1.3 Formulating Abstractions with Higher-Order Procedures. There are a few paraphrases and additional examples added.

The main difference is that the code has been converted from Lisp to Ruby.

The full text of this book and accompanying video lectures can be found at:
<http://swiss.csail.mit.edu/classes/6.001/abelson-sussman-lectures/>

The video lectures are copyright by Hal Abelson and Gerald Jay Sussman. The video lectures, and in turn this document, are licensed under a Creative Commons License.
<http://creativecommons.org/licenses/by-sa/2.0/>



```
def cube(a)
  a * a * a
end
```

Mathematical procedures are, in effect, abstractions that describe compound operations on numbers independent of the particular numbers.

For example, when we define `#cube` we are not talking about the cube of a particular number, but rather about a method for obtaining the cube of any number.

Of course we could get along without ever defining this procedure...

- Procedures are abstractions

```
def cube ( a )  
  a * a * a  
end
```

Mathematical procedures are, in effect, abstractions that describe compound operations on numbers independent of the particular numbers.

For example, when we define `#cube` we are not talking about the cube of a particular number, but rather about a method for obtaining the cube of any number.

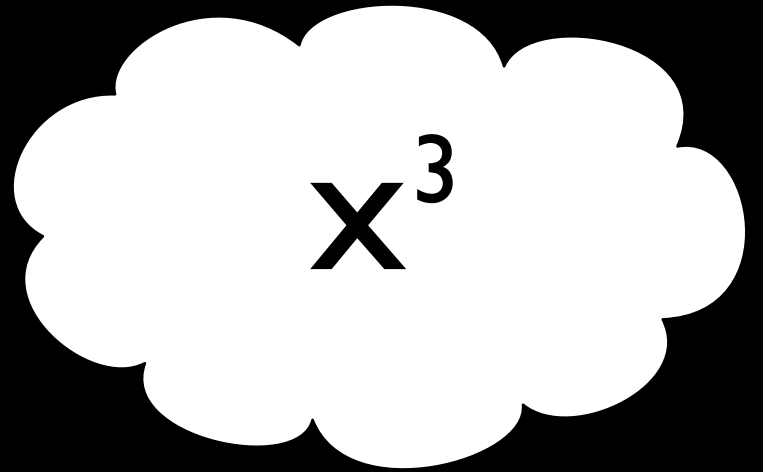
Of course we could get along without ever defining this procedure...

$$\begin{aligned} & (3 * 3 * 3) \\ & (X * X * X) \\ & (Y * Y * Y) \end{aligned}$$

... by always writing expressions such as this and never mentioning cube explicitly. This would place us at a serious disadvantage, forcing us to work always at the level of the particular operations that happen to be primitives in the language (multiplication, in this case) rather than in terms of higher-level operations.

Our programs would be able to compute cubes, but our language would lack the ability to express the concept of cubing.

One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. Procedures provide this ability.



$$\begin{aligned} & (3 * 3 * 3) \\ & (X * X * X) \\ & (Y * Y * Y) \end{aligned}$$

... by always writing expressions such as this and never mentioning cube explicitly. This would place us at a serious disadvantage, forcing us to work always at the level of the particular operations that happen to be primitives in the language (multiplication, in this case) rather than in terms of higher-level operations.

Our programs would be able to compute cubes, but our language would lack the ability to express the concept of cubing.

One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. Procedures provide this ability.

- We need more than numbers for parameters

Yet even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to procedures whose parameters must be numbers.

Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts, we will need to construct procedures that can accept procedures as arguments or return procedures as values.

Procedures that manipulate procedures are called higher-order procedures. This presentation shows how higher-order procedures can serve as a powerful abstraction and vastly increase the expressive power of our language.

- We need more than numbers for parameters
- Patterns indicate concepts

Yet even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to procedures whose parameters must be numbers.

Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts, we will need to construct procedures that can accept procedures as arguments or return procedures as values.

Procedures that manipulate procedures are called higher-order procedures. This presentation shows how higher-order procedures can serve as a powerful abstraction and vastly increase the expressive power of our language.

- We need more than numbers for parameters
- Patterns indicate concepts
- Higher-Order Procedures

Yet even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to procedures whose parameters must be numbers.

Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts, we will need to construct procedures that can accept procedures as arguments or return procedures as values.

Procedures that manipulate procedures are called higher-order procedures. This presentation shows how higher-order procedures can serve as a powerful abstraction and vastly increase the expressive power of our language.

- We need more than numbers for parameters
- Patterns indicate concepts
- Higher-Order Procedures
- Manipulating Procedures = Power

Yet even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to procedures whose parameters must be numbers.

Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts, we will need to construct procedures that can accept procedures as arguments or return procedures as values.

Procedures that manipulate procedures are called higher-order procedures. This presentation shows how higher-order procedures can serve as a powerful abstraction and vastly increase the expressive power of our language.

some examples

Consider the following three procedures.

The first computes the sum of the integers from a through b:

```
def sum_integers(a, b)
  return 0 if a > b
  a + sum_integers((a + 1), b)
end
```

```
sum_integers(1, 10) #=> 55
```

The second computes the sum of the cubes of the integers in the given range:

```
def sum_cubes(a, b)
  return 0 if a > b
  cube(a) + sum_cubes((a + 1), b)
end
```

```
sum_cubes(1, 3) ==> 36
```

The third computes the sum of a sequence of terms in the series:

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} \dots$$

which converges to $\pi/8$ (very slowly)

```
def pi_sum(a, b)
  return 0 if a > b
  (1.0 / ((a + 2) * a)) + (pi_sum((a + 4), b))
end
```

```
pi_sum(1, 1000) * 8 ==> 3.13959265558978
```

a pattern...

```
def sum_integers(a, b)
  return 0 if a > b
  a + sum_integers((a + 1), b)
end
```

```
def sum_cubes(a, b)
  return 0 if a > b
  cube(a) + sum_cubes((a + 1), b)
end
```

```
def pi_sum(a, b)
  return 0 if a > b
  (1.0 / ((a + 2) * a)) + (pi_sum((a + 4), b))
end
```

These three procedures clearly share a common underlying pattern.

They are for the most part identical, differing only in the

- * name of the procedure
- * the function of a used to compute the term to be added, and
- * the function that provides the next value of a.

We could generate each of the procedures by filling in slots in the same template:

a pattern...

```
def sum_integers(a, b)
  return 0 if a > b
  a + sum_integers((a + 1), b)
end
```

```
def sum_cubes(a, b)
  return 0 if a > b
  cube(a) + sum_cubes((a + 1), b)
end
```

```
def pi_sum(a, b)
  return 0 if a > b
  (1.0 / ((a + 2) * a)) + (pi_sum((a + 4), b))
end
```

These three procedures clearly share a common underlying pattern.

They are for the most part identical, differing only in the

- * name of the procedure
- * the function of a used to compute the term to be added, and
- * the function that provides the next value of a.

We could generate each of the procedures by filling in slots in the same template:

a pattern...

```
def sum_integers(a, b)
  return 0 if a > b
  a + sum_integers((a + 1), b)
end
```

```
def sum_cubes(a, b)
  return 0 if a > b
  cube(a) + sum_cubes((a + 1), b)
end
```

```
def pi_sum(a, b)
  return 0 if a > b
  (1.0 / ((a + 2) * a)) + (pi_sum((a + 4), b))
end
```

These three procedures clearly share a common underlying pattern.

They are for the most part identical, differing only in the

- * name of the procedure
- * the function of a used to compute the term to be added, and
- * the function that provides the next value of a.

We could generate each of the procedures by filling in slots in the same template:

a pattern...

```
def sum_integers(a, b)
  return 0 if a > b
  a + sum_integers((a + 1), b)
end
```

```
def sum_cubes(a, b)
  return 0 if a > b
  cube(a) + sum_cubes((a + 1), b)
end
```

```
def pi_sum(a, b)
  return 0 if a > b
  (1.0 / ((a + 2) * a)) + (pi_sum((a + 4), b))
end
```

These three procedures clearly share a common underlying pattern.

They are for the most part identical, differing only in the

- * name of the procedure
- * the function of a used to compute the term to be added, and
- * the function that provides the next value of a.

We could generate each of the procedures by filling in slots in the same template:

template

```
def <name>(a, b)
  return 0 if a > b
  <term>(a) + <name>(<next>(a), b)
end
```

summation

$$\sum_{n=a}^b f(n) = f(a) + \dots + f(b)$$

Mathematicians long ago identified the abstraction of summation of a series and invented "sigma notation," for example to express this concept.

The power of sigma notation is that it allows mathematicians to deal with the concept of summation itself rather than only with particular sums

For example, you can formulate general results about sums that are independent of the particular series being summed.

Similarly, as program designers, we would like our language to be powerful enough so that we can write a procedure that expresses the concept of summation itself rather than only procedures that compute particular sums.

```
def <name>(a, b)
  return 0 if a > b
  <term>(a) + <name>(<next>(a), b)
end
```

We can do this in Ruby by taking the template shown above and transforming the ``slots" into formal parameters:

```
def <name>(a, b)
  return 0 if a > b
  <term>(a) + <name>(<next>(a), b)
end
```

```
def sum(term, a, the_next, b)
  return 0 if a > b
  term.call(a) + sum(term, the_next.call(a), the_next, b)
end
```

We can do this in Ruby by taking the template shown above and transforming the ``slots" into formal parameters:

sum cubes

```
def inc(n)  
  n + 1  
end
```

```
def sum_cubes(a, b)  
  cube = self.method(:cube).to_proc  
  inc   = self.method(:inc).to_proc  
  sum(cube, a, inc, b)  
end
```

```
sum_cubes(1, 3) ==> 36
```

Now to define sum cubes all we have to do is:

Notice here we are transforming the method `:cube` into a Ruby Proc object.

Then we are taking that procedure and passing it as an argument, just like another piece of data.

sum cubes

```
def inc(n)  
  n + 1  
end
```

```
def sum_cubes(a, b)  
  cube = self.method(:cube).to_proc  
  inc = self.method(:inc).to_proc  
  sum(cube, a, inc, b)  
end
```

```
sum_cubes(1, 3) #=> 36
```

Now to define sum cubes all we have to do is:

Notice here we are transforming the method `:cube` into a Ruby Proc object.

Then we are taking that procedure and passing it as an argument, just like another piece of data.

sum integers

```
def identity(x)
```

```
  x
```

```
end
```

```
def sum_integers(a, b)
```

```
  id = self.method(:identity).to_proc
```

```
  inc = self.method(:inc).to_proc
```

```
  sum(id, a, inc, b)
```

```
end
```

```
sum_integers(1, 10) ==> 55
```

π sum

```
def pi_term(x)
  (1.0 / (x * (x+2)))
end

def pi_next(x)
  (x + 4)
end

def pi_sum(a, b)
  term = self.method(:pi_term).to_proc
  nex  = self.method(:pi_next).to_proc
  sum(term, a, nex, b)
end
```

Using these procedures, we can compute an approximation to pi.

Rather than define pi-next and pi-term, it would be more convenient to have a way to directly specify
* ``the procedure that returns its input incremented by 4" and
* ``the procedure that returns the reciprocal of its input times its input plus 2."

We can do this by introducing the special form lambda, which creates procedures.

π sum

```
def pi_term(x)
  (1.0 / (x * (x+2)))
end

def pi_next(x)
  (x + 4)
end

def pi_sum(a, b)
  term = self.method(:pi_term).to_proc
  nex  = self.method(:pi_next).to_proc
  sum(term, a, nex, b)
end

pi_sum(1, 1000) * 8 ==> 3.13959265558978
```

Using these procedures, we can compute an approximation to pi.

Rather than define pi-next and pi-term, it would be more convenient to have a way to directly specify
* ``the procedure that returns its input incremented by 4" and
* ``the procedure that returns the reciprocal of its input times its input plus 2."

We can do this by introducing the special form lambda, which creates procedures.

π sum

```
def pi_term(x)
  (1.0 / (x * (x+2)))
end
```

```
def pi_next(x)
  (x + 4)
end
```

```
def pi_sum(a, b)
  term = self.method(:pi_term).to_proc
  nex  = self.method(:pi_next).to_proc
  sum(term, a, nex, b)
end
```

```
pi_sum(1, 1000) * 8 ==> 3.13959265558978
```

Using these procedures, we can compute an approximation to pi.

Rather than define pi-next and pi-term, it would be more convenient to have a way to directly specify
* ``the procedure that returns its input incremented by 4" and
* ``the procedure that returns the reciprocal of its input times its input plus 2."

We can do this by introducing the special form lambda, which creates procedures.

π sum

```
def pi_term(x)
  (1.0 / (x * (x+2)))
end
```

```
def pi_next(x)
  (x + 4)
end
```

```
def pi_sum(a, b)
  term = self.method(:pi_term).to_proc
  nex = self.method(:pi_next).to_proc
  sum(term, a, nex, b)
end
```

```
pi_sum(1, 1000) * 8 #=> 3.13959265558978
```

Using these procedures, we can compute an approximation to pi.

Rather than define pi-next and pi-term, it would be more convenient to have a way to directly specify
* ``the procedure that returns its input incremented by 4" and
* ``the procedure that returns the reciprocal of its input times its input plus 2."

We can do this by introducing the special form lambda, which creates procedures.

π sum

```
def pi_term(x)
  (1.0 / (x * (x+2)))
end
```

```
def pi_next(x)
  (x + 4)
end
```

```
def pi_sum(a, b)
  term = self.method(:pi_term).to_proc
  nex = self.method(:pi_next).to_proc
  sum(term, a, nex, b)
end
```

```
pi_sum(1, 1000) * 8 #=> 3.13959265558978
```

Using these procedures, we can compute an approximation to pi.

Rather than define pi-next and pi-term, it would be more convenient to have a way to directly specify
* ``the procedure that returns its input incremented by 4" and
* ``the procedure that returns the reciprocal of its input times its input plus 2."

We can do this by introducing the special form lambda, which creates procedures.

λ (lambda)

Using lambda we can describe what we want.

The above examples demonstrate how the ability to pass procedures as arguments significantly enhances the expressive power of our programming language.

We can achieve even more expressive power by creating procedures whose returned values are themselves procedures.

λ (lambda)

```
lambda { |x| (x + 4) }
```

Using lambda we can describe what we want.

The above examples demonstrate how the ability to pass procedures as arguments significantly enhances the expressive power of our programming language.

We can achieve even more expressive power by creating procedures whose returned values are themselves procedures.

λ (lambda)

```
lambda { |x| (1.0 / (x * (x+2))) }
```

```
lambda { |x| (x + 4) }
```

Using lambda we can describe what we want.

The above examples demonstrate how the ability to pass procedures as arguments significantly enhances the expressive power of our programming language.

We can achieve even more expressive power by creating procedures whose returned values are themselves procedures.

λ (lambda)

```
def pi_sum(a, b)
  sum( lambda{ |x| (1.0 / (x * (x+2))) },
      a,
      lambda{ |x| (x + 4) },
      b )
end
```

Using lambda we can describe what we want.

The above examples demonstrate how the ability to pass procedures as arguments significantly enhances the expressive power of our programming language.

We can achieve even more expressive power by creating procedures whose returned values are themselves procedures.

another example

```
def even?(i)
  i % 2 == 0
end
```

```
def filter_evens(list)
  new_list = []
  list.each do |element|
    new_list << element if even?(element)
  end
  new_list
end
```

Lets say we want to filter out the even numbers in a list.

This procedure takes a list and returns a new list containing the even numbers.

Well, later on we may want to filter by odd numbers, or by prime numbers. What we need is a way to express the concept of filtration.

another example

```
def even?(i)
  i % 2 == 0
end
```

```
def filter_evens(list)
  new_list = []
  list.each do |element|
    new_list << element if even?(element)
  end
  new_list
end
```

```
filter_evens( [1,2,3,4,5,6,7,8] ) #=> [2, 4, 6, 8]
```

Lets say we want to filter out the even numbers in a list.

This procedure takes a list and returns a new list containing the even numbers.

Well, later on we may want to filter by odd numbers, or by prime numbers. What we need is a way to express the concept of filtration.

returning procedures

```
def make_filter(predicate)
  lambda do |list|
    new_list = []
    list.each do |element|
      new_list << element if predicate.call(element)
    end
    new_list
  end
end
```

(predicate : Logic something that is affirmed or denied concerning an argument of a proposition.)

Notice that #make_filter returns not just a value, but a procedure. This procedure can then be used on other data.

returning procedures

```
def make_filter(predicate)
  lambda do |list|
    new_list = []
    list.each do |element|
      new_list << element if predicate.call(element)
    end
    new_list
  end
end
```

```
filter_odds = make_filter( lambda{ |i| i % 2 != 0 } )
filter_odds.call(list) #=> [1, 3, 5, 7, 9]
```

(predicate : Logic something that is affirmed or denied concerning an argument of a proposition.)

Notice that #make_filter returns not just a value, but a procedure. This procedure can then be used on other data.

returning procedures

```
require 'facet/integer/ordinal'  
10.ordinal #=> "10th"
```

The power of this is that our filter is not limited to numeric evaluations.

Lets say we want to filter by numbers where the “ordinal” ends in “th” such as 10th.

returning procedures

```
require 'facet/integer/ordinal'
```

```
10.ordinal #=> "10th"
```

```
filter_ths = make_filter(  
  lambda do |i|  
    i.ordinal =~ /th$/ ? true : false  
  end  
)
```

The power of this is that our filter is not limited to numeric evaluations.

Lets say we want to filter by numbers where the “ordinal” ends in “th” such as 10th.

returning procedures

```
require 'facet/integer/ordinal'
```

```
10.ordinal #=> "10th"
```

```
filter_ths = make_filter(  
  lambda do |i|
```

```
    i.ordinal =~ /th$/ ? true : false
```

```
  end
```

```
end
```

```
)
```

```
filter_ths.call(list) #=> [4, 5, 6, 7, 8, 9, 10]
```

The power of this is that our filter is not limited to numeric evaluations.

Lets say we want to filter by numbers where the “ordinal” ends in “th” such as 10th.

wrap-up

- identify abstractions
- abstraction = power
- be appropriate

As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs and to build upon them and generalize them to create more powerful abstractions.

This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts.

The significance of higher-order procedures is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.