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.

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.

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.

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.

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**.

```
; Compute sum of integers between a and b (inclusive)
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a
         (sum-integers (+ a 1) b))))

; Compute sum of cubes of integers between a and b (inclusive)
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (* a a a)
         (sum-cubes (+ a 1) b))))

; Compute pi/4  using a formula
(define (pi-sum a b)
  (if (> a b)
    0
    (+ (/ 1.0 (* a (+ a 2)))
       (pi-sum (+ a 4) b))))
```

These three procedures clearly share a common underlying pattern.

```
(define (⟨sequence⟩ a b)
        (if (> a b)                              ; Check for range condition
            0                                    ; Return 0 if the range has been crossed.
            (+ (⟨sequence⟩ a)                    ; Sum up the a^{th} term of the sequence with
               (⟨sequence⟩ (⟨next⟩ a) b))))      ; Sum of the remaining terms in the range.
```

These three procedures clearly share a common underlying pattern.

```
(define ((sequence) a b)
        (if (> a b)                              ; Check for range condition
            0                                    ; Return 0 if the range has been crossed.
            (+ ((sequence) a)                    ; Sum up the a^{th} term of the sequence with
               ((sequence) ((next) a) b))))      ; Sum of the remaining terms in the range.
```

Mathematicians identified the abstraction of summation of a series and invented the "$\sum$ notation".

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

Sigma notation allows mathematicians to deal with the concept of summation itself rather than only with particular sums.

Thus mathematicians can formulate general results about sums that are independent of the particular series being summed.

```
; Generalizing the concept of summation
; <sequence> denotes the sequence to generate successive terms
; <a> denotes the first index of the sequence
; <b> denotes the last index of the sequence
; <next> denotes the process to get the successor of <a>

(define (sum term a next b)
        (if (> a b)
          0
         (+ (term a)
         (sum term (next a) next b))))

(define (square x) (* x x))

(define (inc x) (+ x 2))

(define (sum-alternate-squares a b)
          (sum square a inc b))
```

```
; Generalizing the concept of summation
; <sequence> denotes the sequence to generate successive terms
; <a> denotes the first index of the sequence
; <b> denotes the last index of the sequence
; <next> denotes the process to get the successor of <a>

(define (sum term a next b)
        (if (> a b)
          0
         (+ (term a)
         (sum term (next a) next b))))

(define (square x) (* x x))

(define (inc x) (+ x 2))

(define (sum-alternate-squares a b)
          (sum square a inc b))
```

Let check using an example

```
> (sum-alternate-squares 1 10)
165
> (+ 1 9 25 49 81)
165
```

(s-a-s 1 10)
→ (+ 1 (s-a-s 3 10))
→ (+ 1 (+ 9 (s-a-s 5 10)))
→ (+ 1 (+ 9 (+ 25 (s-a-s 7 10))))
→ (+ 1 (+ 9 (+ 25 (+ 49 (s-a-s 9 10)))))
→ (+ 1 (+ 9 (+ 25 (+ 49 (+ 81 (s-a-s 11 10))))))
→ (+ 1 (+ 9 (+ 25 (+ 49 (+ 81 0)))))
→ (+ 1 (+ 9 (+ 25 (+ 49 81))))
→ (+ 1 (+ 9 (+ 25 130)))
→ (+ 1 (+ 9 155))
→ (+ 1 164)
→ 165

Notice that sum takes as its arguments the lower and upper bounds *a* and *b* together with the procedures *term* and *next*.

We have the essence of summation concept without concerning ourselves with the function whose terms are being summed up. Once we have sum, we can use it as a building block in formulating further concepts like definite integral.

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \ldots \right] dx$$

This approximation can be expressed as a procedure in the following way.

```
(define (definite-integral f a b dx)
  (define (add-dx x)
          (+ x dx))
          (* (sum f (+ a (/ dx 2.0)) add-dx b) dx)
)

> (definite-integral square 0 1 0.001)
0.33333325000000047
> (definite-integral square 0 1 0.0001)
0.3333333249994335
```

Food for thought

- Can we go one step further and generalize the notion of summation?
- Can we come up with an iterative process to capture summation?

In using sum, it seems awkward to have to define trivial procedures such as *next* just so we can use them as arguments to our higher-order procedure.

In some cases, it would be more convenient to have a way to directly specify "the procedure that returns its input incremented by 2".

The lambda procedures are also useful in creating local variables.

$$f(x, y) = (xy)^2 + (x + y)^2 + (xy)(x + y)$$

which we could also express as

$a = xy$

$b = x + y$

$f(x, y) = a^2 + b^2 + ab$

```
(define (f x y)
  (define (f-helper a b)
    (+ (square a)
       (square b)
       (* a b)))
  (f-helper (+ x y)
            (* x y)))
```

This construct is so useful that there is a special form called *let* to make its use more convenient.
The general form of a let expression is

```
(let ((⟨var1⟩ ⟨exp1⟩)
      (⟨var2⟩ ⟨exp2⟩)
      ...
      (⟨varn⟩ ⟨expn⟩))
  ⟨body⟩)
```

let var_i have the value exp_i to compute *body*.

The first part of the let expression is a list of name-expression pairs. When the let is evaluated, each name is associated with the value of the corresponding expression. The body of the let is evaluated with these names bound as local variables.

A let expression is simply syntactic sugar for the underlying lambda application. No new mechanism is required in the interpreter in order to provide local variables

```
((lambda (⟨var1⟩ ... ⟨varn⟩)
    ⟨body⟩)
 ⟨exp1⟩
 ...
 ⟨expn⟩)
```

The following shows the scope of a variable specified by a let expression is the body of the let.

```
> (define y 10)
> (+ (let ((y 3))
            (* y y)) ; body
    y)    ; Add square of y (inside let) to the y outside the let
19
```

The variables' values are computed outside the let. This matters when the expressions that provide the values for the local variables depend upon variables having the same names as the local variables themselves.

```
> (define x 10)
> (let    ((x 3)
           (y (+ x 2)))
    (* x y))
36
```

Here y is assigned the value of 12. This ensures that the order in which the variables are assigned values in the let is not relevant.

Since the f-helper is a useful abstraction tool and it is primarily used in several cases internally by a procedure for storing temperary/local variables, the let becomes a useful tool in this situation.

One can rewrite the *f* procedure using let in the following manner.

```
(define (f_using_let x y)
  (let ((a (+ x y)) (b (* x y)))
     (+ (square a)
       (square b)
       (* a b)))
)

> (f_using_let 3 4)
277
```

let* is a variant of let in scheme.

In a let expression, the initial values are computed before any of the variables become bound; in a let* expression, the bindings and evaluations are performed sequentially, in the order in which the set of bindings are written.

```
; Difference between let and let*

(define example-let
  (let ((x 2))
    (let ((x 3) (y x))
     (* x y))
  )
)

(define example-letstar
   (let ((x 2))
        (let* ((x 3) (y x))
         (* x y))
   )
)
```

The two procedures only differ at one place. let* sets y's value based on the x accessible within the let* block.

```
> example-let
6
> example-letstar
9
```

```
(define example-letstar
   (let ((x 2))
        (let* ((x 3) (y x))
         (* x y))
   )
)

(define example-letstar2
   (let ((x 2))
        (let* ((y x) (x 3))
         (* x y))
   )
)
```

In the two procedures, the bindings are in different order. The causes y to have two different values in the two procedures.

```
> example-letstar
9
> example-letstar2
6
```

It might seem that let* allows more flexiblility.

It might seem that let* allows more flexiblility.
However, one can implement let* using nested-lets.

```
; Implementing let*
; using nested let

(define letstar-using-let
   (let ((x 2))
        (let ((y x))
         (* x y))
   )
)

(define letstar
   (let* ((x 2) (y x))
         (* x y))
)
```

and achieve the same effect. It is therefore possible that let* is implemented in scheme using let.

The ability to pass procedures as arguments significantly enhances the expressive power of a programming language.

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

```
(define square (λ (x) (* x x)))
(define cube   (λ (x) (* x x x)))
(define average (λ (x y) (/ (+ x y) 2)))
(define foo-1 (λ (f x) (average x (f x))))
(define foo-2 (λ (f) (λ (x) (average x (f x)))))


; Alternatively,
; 1 (define (square x) (* x x))
; 2 (define (cube x) (* x x x))
; 3 (define (average x y) (/ (+ x y) 2))
; 4 (define (foo-1 f x) (average x (f x)))
; 5 (define (foo-2 f) (λ (x) (average x (f x))))
```

foo-1 and foo-2 will have the same output for similar input parameters. However there is crucial difference between the two. foo-2 returns a procedure as an output, whereas foo-1 returns an number as the output.

```
> foo-1
#<procedure:foo-1>
> foo-2
#<procedure:foo-2>
> (foo-1 square)
foo-1: arity mismatch;
 the expected number of arguments does n
  expected: 2
  given: 1
> (foo-2 square)
#<procedure:...dures-as-output.rkt:7:21>
```

Both foo-1 and foo-2 are procedures. However foo-1 expects 2 arguments while foo-2 expects one argument. foo-1 returns a value, whereas foo-2 returns a procedure.

```
> (foo-1 square 4)
10
> ((foo-2 square) 4)
10
> (foo-2 square 4)
foo-2: arity mismatch;
 the expected number of argum
  expected: 1
  given: 2
```

The intuition to construct foo-2 in such a manner comes from lambda calculus.

Lambda calculus only uses functions of a single input.

An ordinary function that requires two inputs, for instance the foo-1 function, can be reworked into an equivalent function foo-2 that accepts a single input, and as output returns another function, that in turn accepts a single input.

This concept is known as **currying** in lambda calculus.

Programming languages impose restrictions on the ways in which computational elements can be manipulated.

In general, a computational element in a programming language is said to have

- *first-class* status, if it can be passed as a parameter, returned from a procedure, assigned into a variable, or stored in a data structure. Simple types such as integers and characters are first class values in most programming languages.
- *second-class* status, if it can be passed as a parameter, but not returned from a procedure or assigned into a variable.
- *third-class* status, if it cannot even be passed as a parameter to a procedure.

In scheme, procedures are given full first-class status. This poses challenges for efficient implementation but the resulting gain in expressive power is enormous.

Subroutines are second-class values in most imperative languages.

Labels (such as destinations for goto statements) are third-class values in most programming languages.

Scheme procedure's aren't really just pieces of code you can execute; they're closures.

A **closure** is a procedure that records what environment it was created in. When you call it, that environment is restored before the actual code is executed.

This ensures that when a procedure executes, it sees the exact same variable bindings that were visible when it was created–it doesn't just remember variable names in its code, it remembers what storage each name referred to when it was created.

Procedures are allowed to remember binding environments even after the expressions that created those environments have been evaluated.

For example, a closure created by a lambda inside a *let* expression will remember the *let*'s variable bindings even after we've exited the *let*.

As long as we have a pointer to the procedure (closure), the bindings it refers to are guaranteed to exist.

```
(let ((count 0))
                    (lambda ()
                       (set! count (* count 1))
                       count))

; The above anonymous lambda expression creates a count and intializes it to zero.
; However after it is created, we do not have a pointer to this procedure
; and we lose access to the procedure.
```

We didn't give a name to the value, so we can't refer to it anymore, and the garbage collector will just reclaim it.

However, if we name the *let* procedure, by binding a new variable my-counter, we use the above *let* expression to create a new environment and procedure.

```
(define foo-counter
                  (let ((count 1))
                     (lambda ()
                       (set! count (+ count 1))
                         count)))
```

This procedure which takes no operands has a name foo-counter. Everytime the foo-counter is called, the value of count is incremented by 1. Since we have a name to access the procedure, we also have access to its bindings. If we call the procedure foo-counter, it will execute in its own "captured" environment (created by the let).

Note that the variable count is created only once. Only the value of the count variable keeps changing.

```
> (foo-counter)
2
> (foo-counter)
3
```

It will increment the binding of count in that environment, and return the result. The environment will continue to exist as long as the procedure does, and will store the latest value until next time foo-counter is called.
In this example, when the foo-counter was first defined its variable was set to 1. So when we made the first call to the procedure foo-counter, it kept track of its environment and incremented the value of *count* from 1 to 2.

```
> (foo-counter)
2
> (foo-counter)
3
```

It will increment the binding of count in that environment, and return the result. The environment will continue to exist as long as the procedure does, and will store the latest value until next time foo-counter is called.
In this example, when the foo-counter was first defined its variable was set to 1. So when we made the first call to the procedure foo-counter, it kept track of its environment and incremented the value of *count* from 1 to 2.

You may notice a resemblance between closures and "objects" in the object-oriented sense.
A closure associates data with a procedure, where an object associates data with multiple procedures.

```
1  #lang scheme
2
3  (define foo-counter
4                  (let ((count 1))
5                     (lambda ()
6                       (set! count (+ count 1))
7                         count)))
8
9
10 (define loo-counter
11                 (lambda ()
12                    (let((count 1))
13                        (set! count (+ count 1))
14                          count)))
```

```
Welcome to DrRacket, version 8.7 [cs].
Language: scheme, with debugging; memory limit: 128 MB.
> (foo-counter)
2
> (foo-counter)
3
> (foo-counter)
4
> (loo-counter)
2
> (loo-counter)
2
> (loo-counter)
2
```

foo-counter has a lambda inside the let, whereas loo-counter has the let inside the lambda. So foo-counter initializes count only once and reuses its values, whereas loo-counter initializes count everytime the lambda procedure is invoked.