define

- Define: names a variable and bin

```
==> (define pi 3.14159)
==> (define radius 10)
```

```
(lambda (<formal parameters>) (<body>))
```

e: a lambda expression that creates an unnamed
ure that squares its argument:

```
(lambda (x) (* x x))
```

```
;; Calculate x-squared
(define (square x) (* x x))
> (square 21)
441
> (square (+ 2 5))
49
> (square (square 3))
81
```

```
(if <predicate> <consequent> <alternative>)
```

ample Absolute value procedure

```
(define (abs x)
  (if (< x 0) (- x) x))
```

```
(cond (<p1> <e1>)
      (<p2> <e2>)
        ...
      (<pn> <en>))
```

```
(cond
  [<p1> <e1>]
  [<p2> <e2>]
    ...
  [<pn> <en>])
```

of expressions `(<p> <e>)` is a clause

- Recursive formulation for x^n
  - $x^0 = 1, x > 0$
  - $x^n = x * x^{n-1}, n > 0$

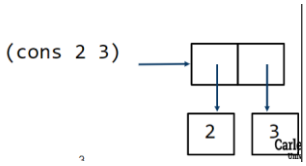
```
(define (power x n)
  (if (= n 0)
      1
      (* x (power x (- n 1)))))
```

```
(power 2 4)
==> (* 2 (power 2 3))
==> (* 2 (* 2 (power 2 2)))
==> (* 2 (* 2 (*  2 (power 2 1))))
==> (* 2 (* 2 (*  2 (* 2 (power 2 0)))))
==> (* 2 (* 2 (*  2 (* 2 1))))
==> (* 2 (* 2 (*  2 2)))
==> (* 2 (* 2 4))
==> (* 2 8)
==> 16
```

Expansion

Contraction

5




```
> (define x (cons 2 3))
> (car x) ; Extraction from 1st cell
2
> (cdr x) ; Extraction from 2nd  cell
3
```

```
(define (list_sum items)
  (if (empty? items)
      0
      (+ (car items)
         (list_sum (cdr items)))))
```

- Retrieve the *n*th item in list `items` (*first item is item 0*)

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

- Summing all number in a list of lists

```
(define (deep-list-sum p)
  (if
   (null? p) 0    ; base case
   (+             ; Sum
    (cond         ; 1st element of the list
          [(list? (car p)) ; inner list
             (deep-list-sum (car p))]
          [(null? (car p)) 0] ; end of list
          [else   (car p)])  ;primitive number
    (deep-list-sum (cdr p)))));with rest oflst
```

```
(define (contains? items target)
  (cond
      [(empty? items) false]
      [(= (car items) target) true]
      [else (contains? (cdr items) target)]))
```


Carleton University

```
> ((lambda (x) (* (sin x) (sin x))) (/ pi 4))
0.5
```

Creating Local Variables

```
(define (roots a b c)
  (cons (/ (+ (- b)
              (sqrt (- (* b b) (* 4 a c))))
           (* 2 a)) ; x = (-b + sqrt(b^2 - 4ac)) / 2a
        (/ (- (- b)
              (sqrt (- (* b b) (* 4 a c))))
           (* 2 a)))) ; x = (-b - sqrt(b^2 - 4ac)) / 2a
```

let Expressions

- Read this as:

let <var_1> have the value <exp_1> and
    <var_2> have the value <exp_2> and
    ...
    <var_n> have the value <exp_n>
in <*body*>

```
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (cons (/ (+ (- b) d) (* 2 a))
          (/ (- (- b) d) (* 2 a)))))
```

- d is now beside the expression that calculates its value
- value of d is calculated above the code where it's used

- A common list operation: apply a transformation to each element in a list, returning the results in a list

```
;; Scale each element in a list by a factor
(define (scale-list items factor)
  (if (null? items)
      '()
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))
```

```
> (scale-list '(1 2 3 4 5) 10)
'(10 20 30 40 50)
```



```
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
```

```
> (accumulate + 0 '(1 2 3 4 5))
15
```

```
> (foldr + 0 '(1 2 3 4))
10
```

```
(define new-count-up
  (let ((counter 0))
    (lambda ()
      (set! counter (+ counter 1))
      counter)))
```

new-count-up

```
> (new-count-up)
1
> (new-count-up)
2
```


begin

- Special form

(begin exp_1 exp_2 .. exp_k)

- Expressions exp_1 exp_2 .. exp_k are evaluated in sequence
- Value of final expression exp_k is returned as the value of the entire begin form

```
(begin (set! counter (- counter 1))
       counter)
```

equal?

- By default, same as eqv?
- For certain datatypes (e.g, strings, pairs, lists), equal? has further specification
- To determine if two lists are equal (contain equal elements arranged in the same order)

```
> (equal? '(1 2 3) '(1 2 3))
#t
> (equal? '(1 2 3) '(4 5 6))
#f
```

eq?

- eq? is true if its arguments refer to the same object
- Used to determine if two symbols are the same; e.g.,

```
> (eq? 'apples 'oranges)
#f
> (eq? 'apples 'apples)
#t
```