Chapter 1

Fundamentals of Higher Order Programming

The Elements of Programming

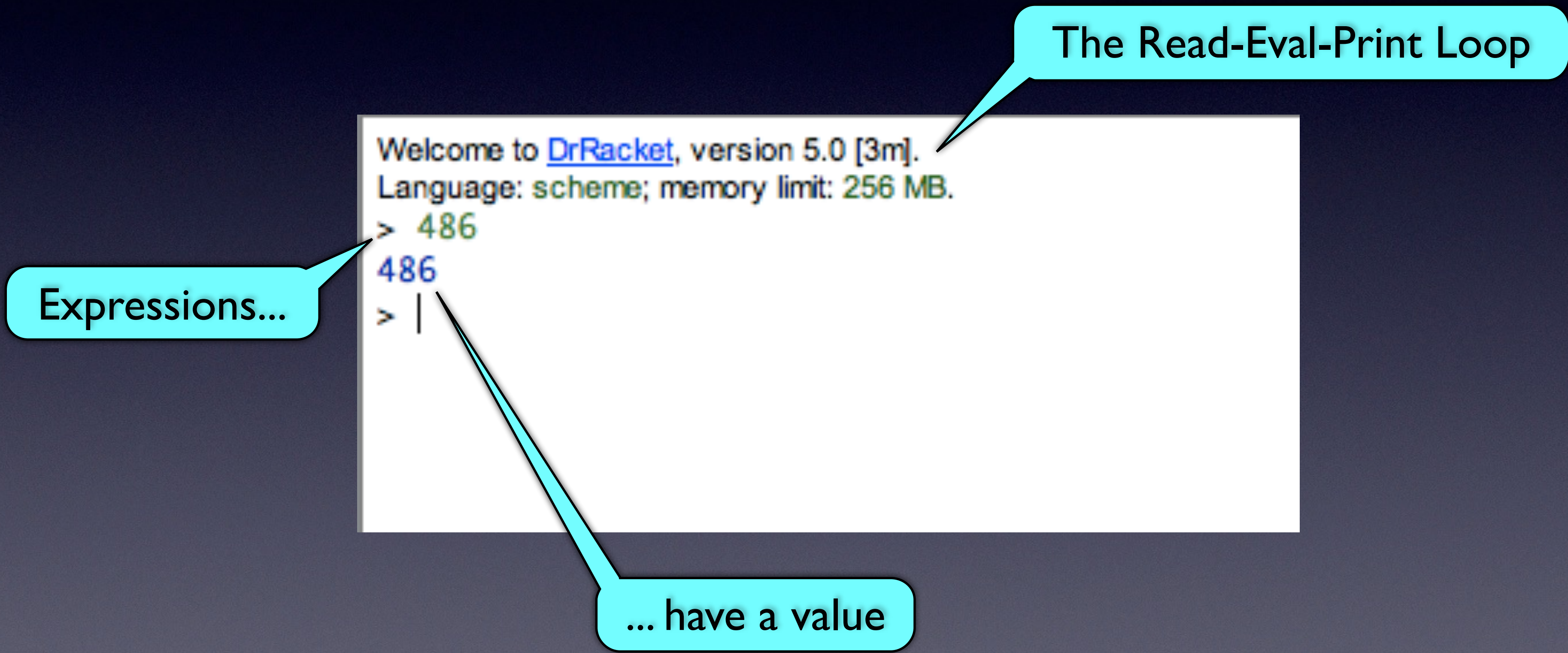
Any powerful language features:

so does Scheme

	data	procedures
primitive		
combinations		
abstraction		

We will see that Scheme uses the same syntax for data and procedures. This is known as homoiconicity.

Expressions, Values & The REPL



Expressions

```
> 4
4
> -5
-5
> (* 5 6 )
30
> (+ 2 4 6 8)
20
> (* 4 (* 5 6))
120
> (* 7 (- 5 4) 8)
56
> (- 6 (/ 12 4) (* 2 (+ 5 6)))
-19
```

Primitive Expressions

Prefix Notation

Combinations

Nested Expressions

Identifiers (aka Variables)

At any point in time, Scheme has access to "an environment"

```
Welcome to DrRacket, version 5.0 [3m].
Language: scheme; memory limit: 256 MB.
> n
⊕ reference to an identifier before its definition: n
> (define n 10)
> n
10
>
```

In the beginning, there is only a "global environment"

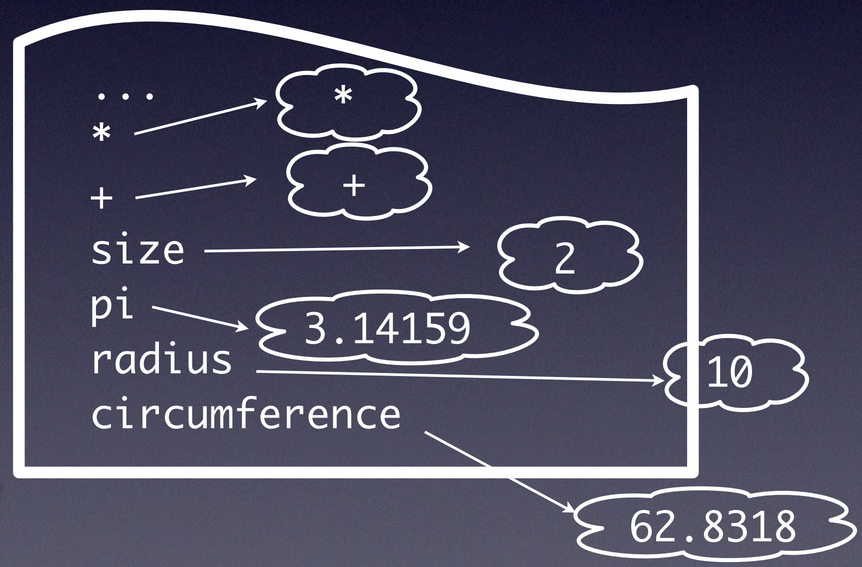
define adds an identifier to the environment

The identifier is bound to a value

```
(define <identifier> <expression>)
```

Examples

```
Welcome to DrRacket, version 5.0 [3m].
Language: scheme; memory limit: 256 MB.
> (define size 2)
> (* 5 size)
10
> (define pi 3.14159)
> (define radius 10)
> (* pi (* radius radius))
314.159
> (define circumference (* 2 pi radius))
> circumference
62.8318
>
```

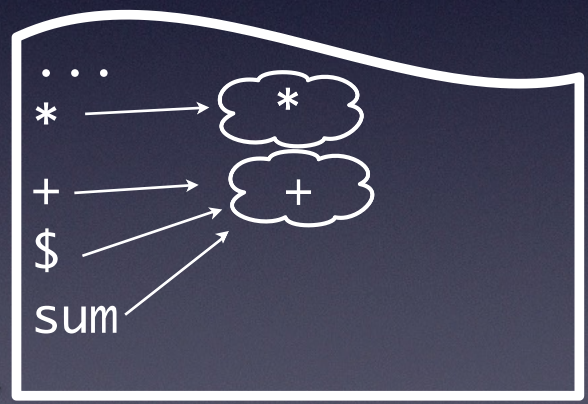


Global environment

Bindings

$, + etc are just identifiers

```
> ($ 4 5)
⊕ reference to an identifier before its definition: $
> (define $ +)
> ($ 4 5)
9
> (define sum +)
sum
> (sum 4 5)
9
```



environment = set of bindings

Scheme's Syntax: S-Expressions

Symbolic Expression

1. An atom, or
2. An combination of the form $(E_1 \; . \; E_2)$ where E_1 and E_2 are S-expressions.

atoms can be numbers, symbols, strings, booleans, …

(x y z) is used as an abbreviation for (x . (y . (z . ‘())))
‘() is pronounced “nil” or “null” or “the empty list”

Scheme

```
(define (fac n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))
```

Common Lisp

```
(defun factorial (x)
   (if (zerop x)
       1
       (* x (factorial (- x 1)))))
```

Evaluation Rules: Version 1

To evaluate an expression:

recursive rule

- numerals evaluate to numbers
- identifiers evaluate to the value of their binding
- combinations:
  - evaluate all the subexpressions in the combination
  - apply the procedure that is the value of the leftmost expression (= the operator) to the arguments that are the values of the other expressions (= the operands)
- some expressions (e.g. define) have a specialized evaluation rule. These are called special forms.

Procedure Definitions

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

To square something, multiply it by itself.

```
(define (<identifier> <formal parameters>) <body>)
```

Procedures (ctd)

```
> (define (square x) (* x x))
> (square 21)
441
> (square (+ 2 5))
49
> (square (square 81))
43046721
> (define (sum-of-squares x y)
    (+ (square x) (square y)))
> (sum-of-squares 3 4)
25
> (define (f a)
    (sum-of-squares (+ a 1) (* a 2)))
> (f 5)
136
>
```

procedure definition

procedure application

building layers of abstraction

The Substitution Model of Evaluation

A "mental" model to explain how procedure application works

```
(f 5) ⇒ (sum-of-squares (+ a 1) (* a 2))      a⟶5
      ⇒ (sum-of-squares (+ 5 1) (* 5 2))
      ⇒ (sum-of-squares 6 10)
      ⇒ (+ (square x) (square y))             x⟶6, y⟶10
      ⇒ (+ (square 6) (square 10))
      ⇒ (+ (* x x) (square 10))               x⟶6
      ⇒ (+ (* 6 6) (square 10))
      ⇒ (+ 36 (square 10))
      ⇒ (+ 36 (* x x))                        x⟶10
      ⇒ (+ 36 (* 10 10))
      ⇒ (+ 36 100)
      ⇒ 136
```

Applicative vs. Normal Order

alternative evaluation model

```
(f 5)  ⇒ (sum-of-squares (+ 5 1) (* 5 2))

       ⇒ (+ (square (+ 5 1)) (square (* 5 2)))

       ⇒ (+ (* (+ 5 1) (+ 5 1)) (square (* 5 2)))

       ⇒ (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))

       ⇒ (+ (* 6 6) (* 10 10))

       ⇒ (+ 36 100)

       ⇒ 136
```

Scheme uses applicative order.

Boolean Values

c.f. truth tables

```
> #t
#t
> #f
#f
> (= 1 1)
#t
> (= 1 2)
#f
> (define true #t)
> true
#t
> (define false #f)
> false
#f
> (and #t #f)
#f
```

predicates

```
> (and (> 5 1) (< 2 5) (= 1 1))
#t
> (or (= 0 1) (> 2 1))
#t
> (not #t)
#f
> (not 1)
#f
> (and 1 2 3)
3
> (or 1 2 3)
1
> (and #f (= "hurray" (/ 1 0)))
#f
> (or #t (/ 1 0))
#t
```

everything is #t, except #f

special forms

case analysis with cond

$$|x| = \begin{cases} x & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -x & \text{if } x<0 \end{cases}$$

```
> (define (abs x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          ((< x 0) (- x))))
> (abs 12)
12
> (abs -3)
3
> (abs 0)
0
```

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

Shorthands

```
> (define (abs x)
    (cond ((< x 0) (- x))
          (else x)))
> (abs -3)
3
> (abs 3)
3
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

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

Special Forms

`cond` `if` `define` `and` `or`

so far

To evaluate a composite expression of the form

(f a1 a2 ... ak)

- if f is a special form, use a dedicated evaluation method
- otherwise, consider f as a procedure application

Case Study: Square Roots

Definition: $\sqrt{x} = y \Leftrightarrow y >= 0 \text{ and } y^2=x$

what is

Procedure:

IF y is guess for $\sqrt{x}$

THEN $\dfrac{y + \dfrac{y}{x}}{2}$ is a better guess

how to

Newton's approximation method

Newton's Iteration Method

```
> (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)
                   x)))
> (define (improve guess x)
    (average guess (/ x guess)))
> (define (average x y)
    (/ (+ x y) 2))
> (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
> (define (sqrt x)
    (sqrt-iter 1.0 x))
> (define (square x)
    (* x x))
> (sqrt 9)
3.00009155413138
```

Iteration is done by ordinary procedure applications

sqrt-iter is a recursive (Eng: re-occur) procedure

procedures are black-box abstractions and can be composed ~"procedural abstraction"

Free vs. Bound Identifiers

A procedure definition binds the formal parameters.
The expression in which the identifier is bound (i.e. the body) is called the scope of the binding.
Unbound identifiers are called free.

good-enough? guess and x are being bound here

```
> (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
```

abs < - square are free

Bound formal parameters are always local to the procedure.

Free identifiers are expected to be bound by the global environment.

Polluted Global Environment

```
> (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)
                   x)))
> (define (improve guess x)
    (average guess (/ x guess)))
> (define (average x y)
    (/ (+ x y) 2))
> (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
> (define (sqrt x)
    (sqrt-iter 1.0 x))
> (define (square x)
    (* x x))
> (sqrt 9)
3.00009155413138
```

The others are "auxiliar procedures"

But everyone can "see" them

Only sqrt is of interest to "users"

Solution: local Definitions

Procedures can have local definitions

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)
                   x)))
  (sqrt-iter 1.0 x))
```

aka block structure

```
(define (<identifier> <formal parameters>)
   <local definitions>
   <body>)
```

Revisited

Lexical Scoping

formal parameters can be free identifiers in the nested definitions

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

Lexical Scoping vs. Dynamic Scoping

Lexical scope: the meaning of a variable depends on the location in the source code and the lexical context. It is defined by the definition of the variable.

most languages

aka static scope

aka static binding

Dynamic scope: the meaning of a variable depends on the execution context that is active when the variable is encountered.

original Lisp, Perl

aka dynamic binding

‘this’ or ‘self’ in OOP

```
void mymethod(y) {
  return this.x + y }
```

```
sub proc1 {
      print “$var\n";
}

sub proc2 {
      local $var = 'local';
      proc1();
}

$var = 'global';

proc1(); # print 'global'
proc2(); # print 'local'
```

Recursion ≠ Recursion

```
(define (fac n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))
```

```
(fac 5)
⇒ (* 5 (fac 4))
⇒ (* 5 (* 4 (fac 3)))
⇒ (* 5 (* 4 (* 3 (fac 2))))
⇒ (* 5 (* 4 (* 3 (* 2 (fac 1)))))
⇒ (* 5 (* 4 (* 3 (* 2 1))))
⇒ (* 5 (* 4 (* 3 2)))
⇒ (* 5 (* 4 6))
⇒ (* 5 24)
⇒ 120
```

linear recursive process

accumulator

```
(define (fac n)
  (fac-iter 1 1 n))
(define (fac-iter product counter max)
  (if (> counter max)
      product
      (fac-iter (* counter product)
                (+ counter 1)
                max)))
```

```
(fac 5)
⇒ (fac-iter 1 1 5)
⇒ (fac-iter 1 2 5)
⇒ (fac-iter 2 3 5)
⇒ (fac-iter 6 4 5)
⇒ (fac-iter 24 5 5)
⇒ (fac-iter 120 6 5)
⇒ 120
```

linear iterative process

Definition

There is a difference between a recursive procedure and a recursive process. An iterative process is a computational process that can be executed with a fixed number of state variables.

```
(define (fac n)
  (fac-iter 1 1 n))
(define (fac-iter product counter max)
  (if (> counter max)
      product
      (fac-iter (* counter product)
                (+ counter 1)
                max)))
```

3 state variables

accumulator

Tail Call Optimisation

A recursive procedure that generates an iterative process is also known as a tail-recursive procedure. Recursion is implemented by means of a runtime stack. Tall-recursive procedures do not need a stack. A compiler that can handle this is said to do tail call optimisation.

```
(define (fac n)
  (fac-iter 1 1 n))

(define (fac-iter product counter max)
  (if (> counter max)
      product
      (fac-iter (* counter product)
                (+ counter 1)
                max)))
```

Tail call optimisation is part of the Scheme's language definition

⇒

```
var product = 1
var counter = 1
var max     = n

label fac-iter
  if (> counter max)
     return product
  else
     product = (* counter product)
     counter = (+ counter 1)
     max     = max
     goto fac-iter
```

3 state variables

Tail Call Optimisation (ctd)

Scheme

```
> (define (happy-printing)
     (display ":-)")
     (happy-printing))

> :-):-):-):-):-):-):-):-):-):-):
-):-):-):-):-):-):-):-):-):-):-):
-):-):-):-):-):-):-):-):-):-):-):
-):-):-):-):-):-):-):-):-):-):-):
-):-):-):-):-):-):-):-):-):-):-):
-):-):-):-):-). . user break
```

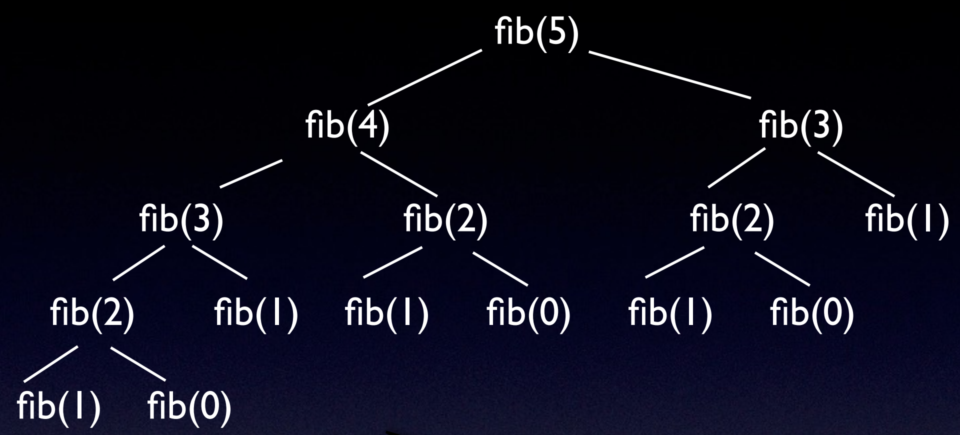
Python

```
>>> def happy_printing():
        print ":-)",
        happy_printing()

>>> happy_printing()
:-) :-) :-) :-) :-) :-) :-) :-) :-) :-)
:-) :-) :-) :-) :-) :-) :-) :-) :-) :-)
:-) :-) :-) :-) :-) :-) :-) :-) :-) :-)
:-) :-) :-) :-) :-) :-) :-) :-) :-) :-)
:-) :-) :-) :-) :-) :-) :-) :-) :-) :-)

. . .

:-) :-) :-) :-) :-) :-) :-) :-) :-) :-)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in
<module>
    happy_printing()
```

Python runs out of stack space

Tree Recursive Processes

```scheme
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```



tree recursive process

accumulators

```scheme
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
(define (fib n)
  (fib-iter 1 0 n))
```

linear iterative process

Exponentiation

linear recursive process

```
(define (exp1 b n)
  (if (= n 0)
      1
      (* b (exp1 b (- n 1)))))
```

linear iterative process

accumulator

```
(define (exp2 b n)
  (exp-iter b n 1))
(define (exp-iter b counter product)
  (if (= counter 0)
      product
      (exp-iter b (- counter 1) (* b product))))
```

logaritmic recursive process

```
(define (exp3 b n)
  (cond ((= n 0) 1)
        ((even? n) (square (exp3 b (/ n 2))))
        (else (* b (exp3 b (- n 1))))))
```

Higher-Order Procedures

A higher-order procedure is a procedure that accepts (a) procedure(s) as argument(s) or one that returns a procedure as the result.

Programming languages put restrictions on the ways elements can be manipulated. Elements with the fewest restrictions are said to have first-class status. Some of the rights and privileges of first-class elements are:

- they may be bound to variables
- they may be passed as arguments to procedures
- they may be returned as results of procedures
- they may be included in data structures

In Scheme, procedures are first-class citizens

Abstracting Common Structure

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))

(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))

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

(define (cube x) (* x x x))

$\frac{1}{1.3} + \frac{1}{5.7} + \frac{1}{9.11} + \ldots$ converges to $\frac{\pi}{8}$

Procedures as Argument

higher-order procedure

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

```
(define (inc n) (+ n 1))
(define (sum-cubes2 a b)
  (sum cube a inc b))

(define (identity x) x)
(define (sum-integers2 a b)
  (sum identity a inc b))

(define (pi-sum2 a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

Example of Reuse

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

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

```
> (integral cube 0 1 0.01)
0.24998750000000042
```

Anonymous Procedures

```
(define (inc n) (+ n 1))
(define (sum-cubes2 a b)
  (sum cube a inc b))

(define (identity x) x)
(define (sum-integers2 a b)
  (sum identity a inc b))

(define (pi-sum2 a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

single usage procedures

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

```
(define (pi-next x) (+ x 4))
```

⇓ ⇓ ⇓

```
(lambda (x) (+ x 4))
```

The procedure of an argument x that adds x to 4

Insight

create 'a procedure' and name it

```
(define (<identifier> <formal parameters>) <body>)
```

$$\Leftrightarrow$$

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

$+$

```
(define <identifier> <expression>)
```

create 'a procedure'

and name it

Examples

```scheme
(define (pi-sum2 a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

⇔

```scheme
(define (pi-sum3 a b)
  (sum (lambda (x)
         (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x)
         (+ x 4))
       b))
```

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

⇔

```scheme
(define (integral f a b dx)
  (* (sum f
          (+ a (/ dx 2.0))
          (lambda (x) (+ x dx))
          b)
     dx))
```

Local Bindings

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

is less clear than:

$$a = (1+xy)$$
$$b = (1-y)$$

$$f(x,y) = xa^2 + yb + ab$$

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

can be used as locally as possible; in any expression

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<varn> <expn>))
   <body>)
```

Insight

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<varn> <expn>))
  <body>)
```

```
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

x is free

⇔

```
((lambda (<var1> ... <varn>)
   <body>)
 <exp1> <exp2> ... <expn>)
```

```
((lambda (x y)
   (* x y))
 3 (+ x 2))
```

x is free

A language construct that is executed by first converting into another (more fundamental) language construct is said to be syntactic sugar.

Variation

```
(let* ((<var1> <exp1>)
       (<var2> <exp2>)
       ...
       (<varn> <expn>))
   <body>)
```

⇔

```
((lambda (<var1>)
   ((lambda (<var2>)
      ...
        ((lambda (<varn>)
           <body>)
         <expn>)
      <exp2>)
   <exp1>)
```

Calculating Fixed-Points

x is a fixed-point of f if and only if f(x) = x

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

for some f, we can approximate x using some initial guess g and calculate f(g), f(f(g)), f(f(f(g))), ...

```
> (fixed-point cos 1.0)
0.7390822985224023
> (fixed-point (lambda (y) (+ (sin y)
                              (cos y)))
               1.0)
1.2587315962971173
```

Improving Convergence

$\sqrt{x} = y \Leftrightarrow y \geq 0$ and $y^2 = x \Leftrightarrow y = x/y$

Hence:
```
(define (sqrt2 x)
  (fixed-point (lambda (y) (/ x y)) 1.0))
```

oscillates between 2 values

But this does not converge! $y_1 \Rightarrow x/y_1 \Rightarrow x/\ x/y_1 = y_1$

take the average of those values

```
(define (sqrt3 x)
  (fixed-point (lambda (y) (average y (/ x y))) 1.0))
```

"average damping"

Making the Essence Explicit

I take a procedure

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

I return a procedure

example

```
> ((average-damp square) 10)
55
```

every idea made explicit

```
(define (sqrt4 x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

reuse all ideas

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y)
                               (/ x (square y))))
               1.0))
```

more neat stuff in the book

$$\sqrt[3]{x} = y \Leftrightarrow y = x/y^2$$

Chapter 1