Chapter 1

only numbers

Fundamentals of Higher-Order Functional Programming

Elements of programming

including Scheme

Every powerful language has three mechanisms for combining simple ideas to form more complex ones:

- **primitive expressions**, which represent the simplest entities the language is concerned with,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

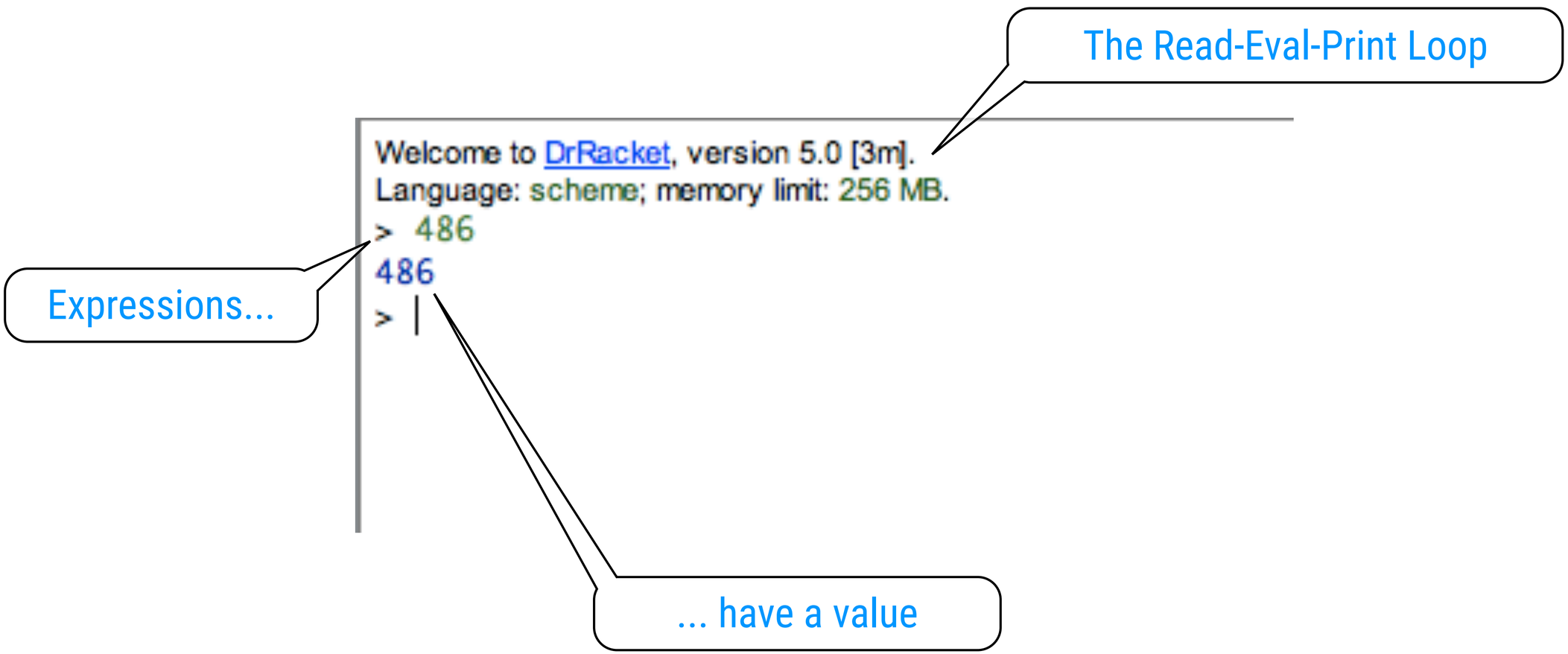
Two kinds of elements: **data** ("stuff" we want to manipulate) and **procedures** (description of the rules for manipulating data)

Elements of programming

	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

Primitive Expressions

Combinations

Prefix Notation

Nested 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
```

Identifiers (aka Variables)

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

In the beginning, there is only a “global 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
>
```

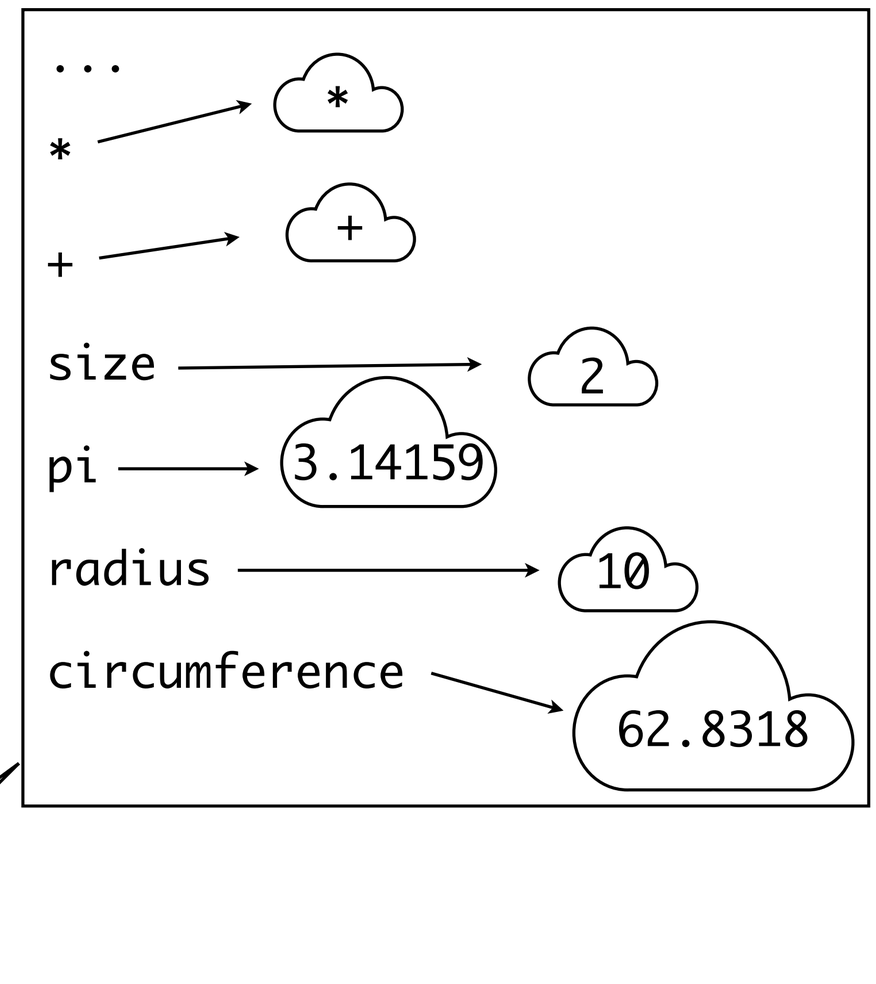
`define` adds an identifier to the environment

The identifier is bound to a value

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

Examples

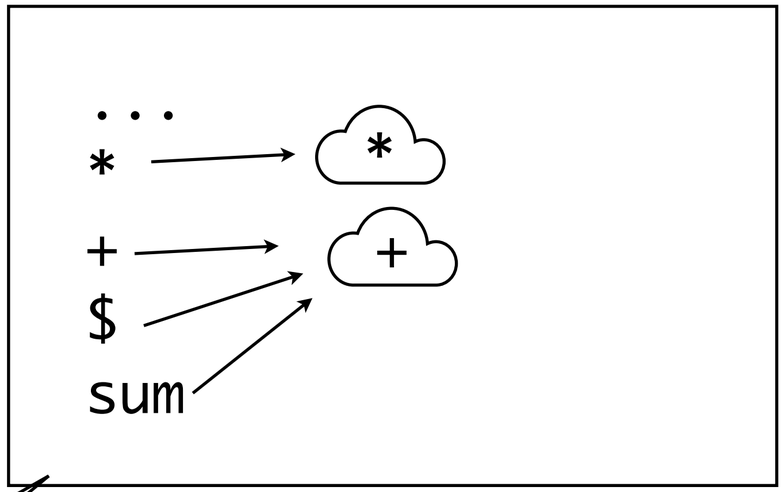
```
Welcome to DrRacket, version 5.0 [3m].
Language: scheme; memory limit: 256 MB.
> (define size 2)
> (* 5 size)
10
> (define pi 3.14159)
> (define radius 10)
> (* pi (* radius radius))
314.159
> (define circumference (* 2 pi radius))
> circumference
62.8318
>
```



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 Syntax = S-Expressions

Symbolic Expression

1. An atom, or
2. A 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”

atom!

Scheme

```
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
```

Common Lisp

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

Anything is Possible (1/2)

Insertion Sort

```
(define (insertion-sort vector <<?)
  (define (>=? x y) (not (<<? x y)))
  (let outer-loop
    ((outer-idx (- (vector-length vector) 2)))
    (let
        ((current (vector-ref vector outer-idx)))
      (vector-set!
       vector
       (let inner-loop
         ((inner-idx (+ 1 outer-idx)))
         (cond
           ((or (>= inner-idx (vector-length vector))
                (>=? (vector-ref vector inner-idx)
                     current))
            (- inner-idx 1))
           (else
            (vector-set! vector (- inner-idx 1)
                                 (vector-ref vector inner-idx))
            (inner-loop (+ inner-idx 1)))))
       current)
      (if (> outer-idx 0)
        (outer-loop (- outer-idx 1))))))
```

Anything is Possible (2/2)

A Scheme Interpreter

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (eval (definition-value exp) env)
    env)
  'ok)

(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))

(define (variable? exp) (symbol? exp))

(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))

(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #f))
```

```
(define (assignment? exp)
  (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))

(define (definition? exp)
  (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)   ; formal parameters
                   (cddr exp)))) ; body

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin seq) (cons 'begin seq))

(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))
```

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

(define (true? x)
  (not (eq? x #f)))
(define (false? x)
  (eq? x #f))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? p)
  (tagged-list? p 'procedure))
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))


(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
  (newline))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))

(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))

(define primitive-procedures
  (list (list 'car car)
        (list 'second cdr)
        (list 'cons cons)
        (list 'null? null?)
        ))
(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))

(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'false #f initial-env)
    (define-variable! 'true #t initial-env)
    initial-env))

(define the-global-environment (setup-environment))

(define input-prompt ";;; M-Eval:")
(define output-prompt ";;; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string)
  (newline))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     'procedure-environment))
      (display object)))

(driver-loop)
```

Evaluation Rules: Version 1

To evaluate an expression:

- numerals evaluate to numbers
- identifiers evaluate to the value of their binding
- combinations:

  recursive rule

  - 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>)
```

Compare with slide 6

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

a ⟶ 5

x ⟶ 6, y ⟶ 10

x ⟶ 6

x ⟶ 10

```
(define (sum-of-squares x y)
    (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

Applicative order: first eval operator + operands, then apply resulting proc to resulting args

Alternative: Normal Order

Scheme uses applicative order

```
(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
```

```
(define (sum-of-squares x y)
    (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

Normal order: only evaluate operands when their value is needed

So far...

The expressive power of the class of procedures that we can define at this point is very **limited**, because we have no way to make tests and to perform different operations depending on the result of a test.

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>))
```

Shorthand: if

```
> (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

so far...

cond

if

define

and

or

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

Newton's approximation method

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

what is

Procedure:

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

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

how to

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 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 “auxiliary procedures” yet everyone can “see” them

Only `sqrt` is of interest to “users”

Solution: Local (i.e. nested) 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))
```

Procedures can have local definitions

a.k.a. block structure

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

Revisited

Lexical Scoping

```
(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))
```

formal parameters can be free identifiers in the nested definitions

Enables simplification (but would hinder reuse)

Lexical Scoping vs. Dynamic Scoping

a.k.a. static binding or static scope

most languages

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.

original Lisp, Perl

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

a.k.a. 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)))
```

nothing happens with result (tail recursion)

```
(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

Procedures and Processes

A procedure is a pattern for the local evolution of a computational process.

recursive procedure ≠ recursive process

What about the global behavior of a process whose local evolution has been specified by a procedure?

Iterative 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. Tail-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 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
```

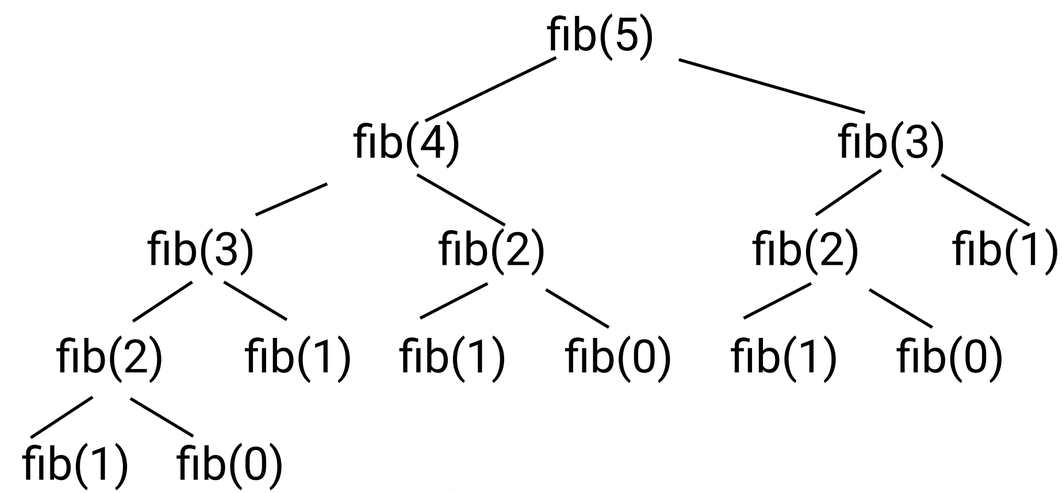
Python

```
>>> 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

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



tree recursive process

accumulators

```
(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)))))
```

Θ(n) steps, Θ(n) space

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))))
```

Θ(n) steps, Θ(1) space

logarithmic recursive process

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

Θ(log n) steps, Θ(log n) space

So far...

Even in numerical processing we would 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.

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))))
```

$$\sum$$

mathematicians long ago identified the abstraction of summation of a series

```
(define (inc n) (+ n 1))
(define (identity x) x)

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

(define (sum-cubes2 a b)
  (sum cube 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

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

formula for numerical approximation of definite integral of function f between limits a and b

```
(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>)
```

⇔

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

+

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

create ‘a procedure’

and name it

Examples

```
(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))
```

⇔

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

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

⇔

```
(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))))
```

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

can be used as locally as possible; in any expression

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

Chapter 1