The Joys of Scheme

Daniel P. Friedman
Computer Science Departmert
Indiana University

XX Conferencia Latinoamericana de Informatica.
PANEL 1994.
Mexico City

Desiderata

- Keep it in your head
  - Simple Syntax.
  - Simple Semantics.
- Freedom of Expression
  - Tinkertoys
  - High Paradigmicity.
  - Dynamically (but Strongly) Typed.
- We shouldn't have to do that!
  - Automatic Memory Management
  - Interactive Program Development

Outline

- Overview of the language
  - Identifying Characteristics
  - Basic Data Types
  - Basic Control Structures
- Simple Program Styles
- Quicksort – An illustration of basic features
  - The algorithm
  - A list implementation
- Advanced Features
  - Syntax Extensions
  - Fixed-Points – Another Example
  - Continuations – Milestones, Devils and Angels
  - Using Continuations – Exception Handling
  - Engines – Amb (simple version)
  - Numeric Multiple Integration

An Overview of Scheme

Identifying Characteristics

- Everything is data.
  - Procedures
  - Continuations
- All objects have indefinite extent.
  - Procedures.
- Dynamic, Strong Typing
- List-based syntax.

Syntax

$$
\begin{array}{lcl}
\textit{expression} & \rightarrow & \textit{literal} \\
 & \rightarrow & \textit{variable} \\
 & \rightarrow & (\textit{expression} \ \ldots) \\
 & \rightarrow & (\mathsf{lambda}\ (\textit{variable} \ \ldots)\ \textit{expression} \ \ldots) \\
 & \rightarrow & (\mathsf{set!}\ \textit{variable expression}) \\
 & \rightarrow & (\mathsf{if}\ \textit{expression expression expression})
\end{array}
$$

- There are also additional *special forms.*
- They denote other language constructs.
- But are built from this small set.

Writing Recursive Programs

```
(define *
  (lambda (n m)
    (cond
      ((zero? n) ?)
      (else (?? (* (sub1 n) m))))))
```

- What is the value of ??
- We reason as follows:
- (* 5 8) $\rightarrow$ 40
- (* 4 8) $\rightarrow$ 32
- How do we fill in ?? to turn 32 into 40?
- Simple, we add 8 to it.
- Lucky for us, m is 8.
- What is the value of ?
- It must be the identity of + (that is 0).

Multiplication

```
(define *
  (lambda (n m)
    (cond
      ((zero? n) 0)
      (else (+ m (* (sub1 n) m))))))
```

(* 5 8) → 40

You're not convinced?

```
(define factorial
  (lambda (n)
    (cond
      ((zero? n) ?)
      (else (?? (factorial (sub1 n)))))))
```

- What is the value of ??
- We reason as follows:
- (factorial 5) → 120
- (factorial 4) → 24
- How do we fill in ?? to turn 24 into 120?
- Simple, we multiply it by 5.
- Lucky for us, n is 5.
- What is the value of ?
- It must be the identity of ∗ (that is 1)

Factorial

```
(define factorial
  (lambda (n)
    (cond
      ((zero? n) 1)
      (else (* n (factorial (sub1 n)))))))
```

(factorial 5) → 120

You're still not convinced?

```
(define power-set
  (lambda (set)
    (cond
      ((null? set) ?)
      (else (?? (power-set (cdr set)))))))
```

- What is the value of ??
- We reason as follows:
- (power-set '(a b c)) $\rightarrow$
  ((a b c) (a b) (a c) (a) (b c) (b) (c) ())
- (power-set '(b c)) $\rightarrow$ ((b c) (b) (c) ())
- How do we fill in ?? to turn "little set" into "big set"?
- Simple, We form the union of adding a to each element of the little set and the little set itself.
- Lucky for us, (car set) is a.
- What is the value of ?: by definition (()).

Power Set

```
(define power-set
  (lambda (set)
    (cond
      ((null? set) '(()))
      (else (extend (car set) (power-set (cdr set)))))))
(define extend
  (lambda (item power-set)
    (append
      (map (lambda (set) (cons item set)) power-set)
      power-set)))
```

Simple Programs

```
(define factorial-aps
  (lambda (n a)
    (cond
      ((zero? n) a)
      (else (factorial-aps (sub1 n) (* n a))))))
(define factorial
  (lambda (n)
    (factorial-aps n 1)))

(define factorial-cps
  (lambda (n k)
    (cond
      ((zero? n) (k 1))
      (else (factorial-cps (sub1 n)
               (lambda (v) (k (* v n))))))))
(define factorial
  (lambda (n)
    (factorial-cps n (lambda (x) x))))
```

Factorial using letrec

```
(define factorial
  (letrec
    ((factoral-aps
       (lambda (n a)
         (cond
           ((zero? n) a)
           (else (factorial-aps (sub1 n) (* n a)))))))
    (lambda (n)
      (factorial-aps n 1))))
```

```
(define factorial
  (letrec
    ((factorial-cps
       (lambda (n k)
         (cond
           ((zero? n) (k 1))
           (else (factorial-cps (sub1 n)
                   (lambda (v) (k (* n v)))))))))
    (lambda (n)
      (factorial-cps n (lambda (v) v)))))
```

Basic Data Types

Atomic Data Types

- Numbers: 9, 10.9, 33e10
- Characters: #\a, #\newline
- Symbols: video, cd-player, radio?

Compound Objects

- Lists:

  Constructors: '(), cons, list

  Accessors/Mutators: car, cdr, set-car!, set-cdr!

  Predicates: null?, pair?, list?

- Vectors:

  Constructor: vector

  Accessor/Mutator: vector-ref, vector-set!

  Predicate: vector?

- Strings:

  Constructor: string

  Accessor/Mutator: string-ref, string-set!

  Predicate: string?

Unprintable Data Types

- Procedures

  Constructor: (lambda *formals body*)

  Accessor: (*operator operands* ...)   (Application).

  Predicate: procedure?

- Continuations

  Constructor: call/cc

  Accessor: *Application*

  Predicate: procedure?

Basic Control Structures

- Conditionals:
  - (if *test-exp then-exp else-exp*)
  - (cond (*test-exp exp*) ...)
- Sequencing: (begin $E_0 \ldots E_n$)
- Looping and Iteration: By Recursion.

An Illustration: Quicksort

Problem: Sort a given set, say X, of numbers in ascending order. For example:

$$7\ 3\ 8\ 9\ 1$$

.

Boundary Condition : If empty set, sorted version is the empty sequence.

Step 1 Choose a pivot: n s.t. $n \in X$.
$\boxed{7}$ 3 8 9 1

Step 2 Partition the list: $\langle A, B \rangle$ s.t.
$A = \{x \in X \mid x < n\}$ and $B = \{x \in X \mid x > n\}$
$\boxed{7}$ $\boxed{3\ 1}$ $\boxed{8\ 9}$

Step 3 Sort partitions and join them: $A' + +[n] + +B'$, where A' and B' are sorted versions of A and B.
1 3 7 8 9

Represent sets as lists

```
(define quicksort
  (lambda (ls)
    (cond
      ((null? ls) '())
      (else (let ((pivot (car ls)))
              (append
                (quicksort (lower-partition pivot (cdr ls)))
                (list pivot)
                (quicksort (upper-partition pivot (cdr ls))))))))))
```

- cond is a conditional.
- let is a binding construct
- append joins two lists.

```
(define lower-partition
  (lambda (pivot ls)
    (filter-in (lambda (x) (> pivot x)) ls)))

(define upper-partition
  (lambda (pivot ls)
    (filter-in (lambda (x) (< pivot x)) ls)))

(define filter-in
  (lambda (test ls)
    (cond
      ((null? ls) '())
      ((test (car ls)) (cons (car ls) (filter-in test (cdr ls))))
      (else (filter-in test (cdr ls)))))))
```

Using Continuation Passing Style

```
(define quicksort
  (lambda (ls)
    (cond
      ((null? ls) '())
      (else (let ((pivot (car ls)))
              (partition pivot (cdr ls)
                (lambda (lower-partition upper-partition)
                  (append
                    (quicksort lower-partition)
                    (list pivot)
                    (quicksort upper-partition))))))))))
```

```
(define partition
  (lambda (pivot ls k)
    (cond
      ((null? ls) (k '() '()))
      (else (partition pivot (cdr ls)
              (lambda (lower-partition upper-partition)
                (if (< (car ls) pivot)
                    (k (cons (car ls) lower-partition)
                       upper-partition)
                    (k lower-partition
                       (cons (car ls) upper-partition))))))))))
```

Semantics of some special forms

... In terms of the basic syntax.

```
(let ((v e) ...) body0 body1 ...) ⇒
((lambda (v ...) body0 body1 ...) e ...)

(block (v ...) body0 body1 ...) ⇒
(let ((v '_) ...) body0 body1 ...)

(letrec ((v e) ...) body0 body1 ...) ⇒
(block (v ...) (set! v e) ... body0 body1 ...)

(begin e0 e1 ...) ⇒ ((lambda () e0 e1 ...))

(while test-exp e0 e1 ...) ⇒
(letrec
  ((loop (lambda ()
           (if test-exp (begin e0 e1 ... (loop))))))
  (loop))
```

Fixed-Points – Another Example

Fixed-point algorithms are fairly common

- Numerical Algorithms.
- Set Equations.
- Program Analyses *etc.*

Repeat a certain computation until changes in the solution are within a certain bound. For example, let us compute the value of π.

$$\pi = 4tan^{-1}(1)$$

$$tan^{-1}(1) = 1 - 1/3 + 1/5 - 1/7 + 1/9 \ldots$$

```
(define pi
  (letrec-equations
    ((x 0 (+ x (* (expt -1 n) (/ 1.0 (add1 (* 2 n))))) ≈)
     (n 0 (add1 n) (lambda (x y) #t)))
    (* 4 x)))
```

In such cases, one usually begins with an initial solution and then iterates.

- letrec-equations is a special form!
- It is not in standard Scheme.
- It is a great abstraction for a fairly common style of programming.
- How general is this abstraction?

Set equations are useful in many contexts. For example:

The Philosophers Party:

- A philosopher will attend the party iff his friends will.
- Find a suitable guest-list.

```
(define guest-list
  (lambda (inits)
    (letrec-equations
      ((guests inits (union guests (friends guests)) ≡))
      guests)))
```

```
(guest-list '(aristotle)) →
```

- We can add letrec-equations to Scheme.
- Macros, or syntax extensions, allow syntactic forms to be added to the language.
- Macros are not yet a part of standard Scheme, but will soon be.

```
(letrec-equations ((var init exp binary-test) ...)
  body0 body1 ...)
⇒
(letrec
  ((loop (lambda (var ...)
           (if (and (binary-test var exp) ...)
               (begin body0 body1 ...)
               (loop exp ...)))))
  (loop init ...))
```

```
(define friends
  (let ((lookup-list '((aristotle plato alexander)
                       (plato socrates)
                       (socrates)
                       (alexander ptolemy)
                       (homer virgil sophocles)
                       (virgil dante horace)
                       (dante petrach)
                       (petrach sidney shakespeare))))
    (lambda (guests)
      (map-union (lambda (g)
                   (let ((ans (assq g lookup-list)))
                     (if ans (cdr ans) '())))
                 guests))))
```

Continuations

- A generalized Goto.
- A continuation is a procedure that represents the "rest of the computation".
- A continuation can be captured.
- If a continuation is restored, it resets the computation back to the point the continuation was captured.
- Primary Uses:
  - Non-standard control constructs: coroutines.
  - Exception Handling
  - Backtracking, as in prolog.

Escaping Procedures

Procedures that abort when applied. Examples:

$((\lambda$* (v) $(+$ 3 v)) 4)	$\rightarrow$	7
$(+$ $((\lambda$* (v) $(+$ 3 v)) 4) 9)	$\rightarrow$	7
(f $((\lambda$* (v) $(+$ 3 v)) 4))	$\rightarrow$	7 (For any f)
(f $((\lambda$* (v) (g $(+$ 3 v))) 4))	$\rightarrow$	(g 7)

Call/cc—Basics

(cons 5 (+ 6 7))

```
(call/cc                              k=(λ* (v) v)
  (lambda (k)
    (k (cons 5 (+ 6 7)))))
```

```
((call/cc                   k=(λ* (v) (v 5 (+ 6 7)))
   (lambda (k) (k cons)))
 5 (+ 6 7))
```

```
(cons                    k=(λ* (v) (cons v (+ 6 7)))
  (call/cc
    (lambda (k) (k 5)))
  (+ 6 7))
```

```
(cons 5                                k=(λ* (v) (cons 5 v))
  (call/cc
    (lambda (k) (k (+ 6 7)))))
```

```
(cons 5                                k=(λ* (v) (cons 5 (v 6 7)))
  ((call/cc
     (lambda (k) (k +)))
   6 7))
```

```
(cons 5                                k=(λ* (v) (cons 5 (+ v 7)))
  (+ (call/cc
        (lambda (k) (k 6)))
      7))
```

Milestones, Devils and Angels

Milestone: An important event—worth getting back to.

Devil: Takes you back to your last milestone in exchange for your soul.

Angel: Undoes the work of the devil.

- The devil's job is to keep the computation from happening: Errors and exceptions.
- A milestone is a place where the computation goes back to recover.
- An angel causes the computation to recover.

```
(define mda
  (lambda ()
    (cond
      ((& 'a (= (& 'b (milestone 3)) 4))
       (begin (writeln "here!") (& 'c (angel 'all-is-well))))
      ((& 'd (= (& 'e (milestone 4)) 5))
       (& 'f (angel (& 'g (devil 4)))))
      (else (begin (writeln "made it to") (& 'h (devil 5)))))))
```

```
In &b : 3
In &a : #f
In &e : 4
In &d : #f
made it to
In &e : 5
In &d : #t
In &b : 4
In &a : #t
here!
In &g : all-is-well
In &h : all-is-well
all-is-well
```

```
(define *past* '())
(define *future* '())

(push! s v) ⇒ (set! s (cons v s))

(pop! s) ⇒ (if (null? s)
                (error "Cannot pop empty stack")
                (let ((v (car s)))
                  (set! s (cdr s))
                  v))
```

```
(define milestone
  (lambda (v)
    (call/cc
      (lambda (k)
        (push! *past* k)
        v))))

(define devil
  (lambda (v)
    (call/cc
      (lambda (k)
        (push! *future* k)
        ((pop! *past*) v)))))

(define angel
  (lambda (v)
    ((pop! *future*) v)))
```

Using Continuations—Exception Handling

We want to control how errors are handled.

```
(define div-3
  (lambda (x)
    (if (= x 0) (raise 'divide-by-zero) (/ 3 x))))

(define foo
  (lambda (n)
    (div-3 n)))

(define test
  (with-handler ((divide-by-zero (lambda (exn) #f)))
     (foo 0))) ⇒ #f
```

- raise raises the exception.
- with-handler traps it.

Handlers have *fluid scope* i.e, the last handler established at the time the error occured must be chosen.

- Stack of handlers.
- Handlers should know where to resume execution.

```
(define *handlers* '())

(with-handler ((name handler)) body0 body1 ... )
⇒
(call/cc
  (lambda (k)
    (fluid-let ((*handlers* (cons (list 'name k handler)
                                  *handlers*)))
      body0 body1 ... )))
```

```
(define raise
  (lambda (exn)
    (let ((res (assq exn *handlers*)))
      (cond
        ((pair? res)
         (apply
           (lambda (name k handler)
             (k (handler exn)))
           res))
        (else (error "Exception is not handled" exn))))))
```

Engines—Amb (Simple Version)

Engines: An abstraction of asynchronous control

Amb: Return the value of the expression that returns first.

```
> (amb (factorial 20) (factorial 10))
(2432902008176640000 . 6)
> (amb (factorial 20) (factorial 10))
(3628800 . 2)
> (amb (factorial 20) (factorial 10))
(3628800 . 2)
> (amb (factorial 20) (factorial 10))
(3628800 . 3)
> (amb (factorial 20) (factorial 10))
(2432902008176640000 . 11)
```

- (an-eng ticks success failure)
- ticks is an integer.
- success is a 2-argument procedure.
- Its first argument is the value of the original argument to engine.
- Its second argument is the unused ticks.
- failure is a 1-argument procedure.
- Its only argument is a new engine.
- If it is invoked, it starts where the old one quit.

(amb e1 e2) $\Rightarrow$ (amb/engines (engine e1) (engine e2))

```
(define amb/engines
  (lambda (engine1 engine2)
    (engine1 (random 20) cons
      (lambda (engine1*)
        (engine2 (random 20) cons
          (lambda (engine2*)
            (amb/engines engine1* engine2*)))))))
```

What if cons is replaced by (lambda (value ticks) value) ?
What if cons is replaced by (lambda (value ticks) ticks) ?

Gaussian Quadrature

- 5-point Gaussian Quadrature algorithm for finding definite integrals.
- The algorithm is exact to the 9th degree, modulo the inexactness of the coefficients used.
- Scheme's elegance is brought out, as usual, by `lambda`
- The following algorithm computes definite integrals over the interval $[-1.0, +1.0]$, so in general, a translation of the interval is needed.
- Any ordinary language like C or Pascal would require us to define a "help" function defined on the interval $[-1, +1]$, but with Scheme's support of anonymous procedures via `lambda`, this isn't necessary.
- The substitution is done "on the fly".

5-point Gaussian quadrature. Exact to the 9^{th} degree.

```
(define int
  (let ((roots '( 0.9061798459 ;; roots of 5th Legendre poly
                  0.5384693101
                  0.0000000000
                 -0.5384693101
                 -0.9061798459))
        (coeffs '(0.2369268850 ;; coeffs for its respective root
                  0.4786286705
                  0.5688888889
                  0.4786286705
                  0.2369268850)))
```

```
(let ((int-from-neg-1-to-1
        (lambda (f)
          (letrec
            ((loop
               (lambda (coeffs roots)
                 (cond
                   ((null? coeffs) 0.0)
                   (else
                     (+ (* (car coeffs) (f (car roots)))
                        (loop (cdr coeffs) (cdr roots)))))))))
            (loop coeffs roots)))))
  (lambda (f limit-pair)
    (int-from-neg-1-to-1
      (translate f limit-pair (cons -1 1)))))))
```

```
(define translate
  (lambda (f limit-pair1 limit-pair2)
    (let ((from1 (car limit-pair1))
          (to1 (cdr limit-pair1))
          (from2 (car limit-pair2))
          (to2 (cdr limit-pair2)))
      (let ((m (/ (- to1 from1) (- to2 from2))))
        (lambda (x)
          (* m (f (+ (* m (- x from2)) from1))))))))
```

$\int_0^1 x^2 dx = 1/3$

```
> (int (lambda (x) (* x x)) (cons 0.0 1.0))
0.3333333333044613
```

To compute a double integral we need to make sure we get the variables assigned correctly:

$\int_1^2 \int_3^4 xy.dy.dx = 21/4$

```
> (int
    (lambda (x)
      (int
        (lambda (y) (* x y))
        (cons 3.0 4.0)))
    (cons 1.0 2.0))
5.249999999475
```

But with Scheme's ability to curry λ's we can surely do better:

```
> (define f (lambda (x) (lambda (y) (* x y))))
> (int (lambda (x)
         (int (lambda (y) ((f x) y))
           (cons 3.0 4.0)))
     (cons 1.0 2.0))
5.249999999475
```

We can now define double-int as follows:

```
(define double-int
  (lambda (f limit-pair1 limit-pair2)
    (int (lambda (x)
           (int (lambda (y) ((f x) y))
             limit-pair2))
      limit-pair1)))
```

```
> (double-int
    (lambda (x) (lambda (y) (* x y)))
    '(1.0 . 2.0) '(3.0 . 4.0))
5.249999999475
```

We can write an integrator that takes multiple integrations.

```
(define n-int
  (lambda (f limit-pairs)
    (cond
      ((null? limit-pairs) f)
      (else (int (lambda (x)
                   (n-int (f x) (cdr limit-pairs)))
              (car limit-pairs))))))
> (n-int
    (lambda (x) (lambda (y) (* x y)))
    '((1.0 . 2.0) (3.0 . 4.0)))
```

We can abstract over int, allowing any integrator.

```
(define n-int-maker
  (lambda (int)
    (letrec
      ((n-int
        (lambda (f limit-pairs)
          (cond
            ((null? limit-pairs) f)
            (else (int (lambda (x)
                         (n-int (f x) (cdr limit-pairs)))
                       (car limit-pairs)))))))
      n-int)))

(define gaussian-quadrature-n-int (n-int-maker int))
```

Or we can associate a different integration method with each limit pair by forming a new data structure that has both the integrator and its limit pair.

```
(define super-n-int
  (lambda (f int-limits-pairs)
    (cond
      ((null? int-limits-pairs) f)
      (else ((car (car int-limits-pairs))
             (lambda (x)
               (super-n-int (f x) (cdr int-limits-pairs)))
             (cdr (car int-limits-pairs)))))))

> (super-n-int
    (lambda (x) (lambda (y) (* x y)))
    (list (cons int '(1.0 . 2.0)) (cons int '(3.0 . 4.0))))
```

The Simplicity of Scheme

- variable reference, lambda, application
- set!
- if
- call/cc
- engines
- Primitives on data structures
- define
- $\Rightarrow$

Why Scheme?

- Paradigmatic Considerations
  - Simplicity
  - Minimality
  - Flexibility
  - Wide range of concepts can be realized with just a few constructs.
- Technical Considerations
  - Interactive Programming.
  - Automatic Memory Management and Type Safety
  - Implementations on a wide variety of platforms
- Why not?