An Introduction to Scheme

Desiderata

- Keep it in your head
  - Simple Syntax.
  - Simple Semantics.
- Freedom of Expression
  - Tinkertoys
  - High Paradigmacity.
  - Dynamically (but Strongly) Typed.
- I 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
- 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)
- Closing Remarks

An Overview of Scheme

Identifying Characteristics

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

Syntax

$$\begin{array}{lcl} \textit{s-expression} & \rightarrow & \textit{literals} \\ & \rightarrow & \textit{symbols} \\ & \rightarrow & (\textit{s-expression} \ldots) \\ & \rightarrow & {}'\textit{s-expression} \end{array}$$

- Some s-expressions are *special forms*.
- They denote specific language constructs.

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 $E_0 \ldots E_n$)
- Side-effects: (set! v E)
- 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.
  - Tests clauses in sequence.
  - Returns value of the corresponding expression.
- let is a binding construct
  - Sets up variables on the left
  - Provides them with values using expressions on the right.
- append joins two lists.

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

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

(define filter
  (lambda (predicate ls)
    (letrec ((loop (lambda (ls)
                     (cond
                       ((null? ls) '())
                       ((predicate (car ls)) (cons (car ls) (loop (cdr ls))))
                       (else (filter predicate (cdr ls)))))))
      (loop ls))))
```

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 ...))
```

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))))) approx=)
     (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
  (letrec-equations
    ((guests '(aristotle) (union guests (friends guests)) set-equal?)
    guests)))
```

- 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)
                       (horace livy)
                       (no-hun atilla))))
    (lambda (guests)
      (apply union
        (lambda (p)
          (let ((ans (assq p lookup-list)))
            (if ans
              (cdr ans)
              '())))))))
```

Continuations

- A generalized Goto.
- A continuation 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 such as coroutines.
  - Exception Handling
  - Backtracking, as in prolog.

Escaping Procedures

Procedures that abort when applied. Examples:

$$((\lambda^* \ (v) \ (+ \ 3 \ v)) \ 4) \quad \rightarrow \quad 7$$

$$(+ \ ((\lambda^* \ (v) \ (+ \ 3 \ v)) \ 4) \ 9) \quad \rightarrow \quad 7$$

$$(f \ ((\lambda^* \ (v) \ (+ \ 3 \ v)) \ 4)) \quad \rightarrow \quad 7 \text{ (For any f)}$$

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 devils 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.

An example:

```
(define mda
  (lambda ()
    (cond
      ((= (milestone 3) 4) (begin (display "here!")
                                  (angel 'all-is-well)))
      ((= (milestone 4) 5) (angel (devil 4)))
      (else (begin (display "made it to ")
                   (devil 5))))))
```

(mda) → made it to here!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 test
  (with-handler ((divide-by-zero (lambda (exn) #f)))
    (div-3 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)) body)
⇒
(call/cc (lambda (k)
          (fluid-let ((*handlers* (cons (list 'name k handler)
                                        *handlers*)))
            body)))
```

```
(define raise
  (lambda (exn)
    (let ((res (assq exn *handlers*)))
      (if res
          (let ((where-to-go (cadr res))
                (what-to-do (caddr res)))
            (where-to-go (what-to-do exn)))
          (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)
```

```
(amb exp1 exp2)
⇒
(amb/engines
   (make-engine (lambda () exp1))
   (make-engine (lambda () exp2)))


(define amb/engines
  (lambda (e1 e2)
    (e1 (random 20) cons
      (lambda (e1*)
        (e2 (random 20) cons
          (lambda (e2*)
            (amb/engines e1* e2*))))))))
```

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?