Scheme in Introductory CS Courses

John E. Howland
Computer Science Department
Trinity University
715 Stadium Drive
San Antonio, Texas  78212-7200

Voice: (210) 736-7480
Fax: (210) 736-7477
Internet: jhowland@ariel.cs.trinity.edu
Interactive

Scheme systems support interactive program development and debugging. Many Scheme implementations also have compilers.

Simple List Syntax

Lists start and end with "(" and ")" characters. List items are separated by one or more white-space characters. List items may lists.

Same Notation to Describe Procedures and Data

(+ 2 3)

(* (+ 4 5) (= 3 2))

(define square
  (lambda (x) (* x x)))

(define people
  '(((Howland (professor computer science))
     (Clinton (president United States))
     (Sigle (professor computer science)))))

(length people) => 3
First Class Objects

All Scheme objects are first class, that is, there are no restrictions on how they are treated or manipulated. For example, Functions may be passed as arguments to functions and returned as values of functions.

Notation Based on Lambda Calculus

(define plus
  (lambda (x y) (+ x y)))

(define plus-c
  (lambda (x)
    (lambda (y) (+ x y))))

(define plus-4
  (plus-c 4))

(plus-4 5) ==> 9

Reason About Programs

Since Scheme is based on a mathematical notation, it has the potential of being easier to reason about the meaning and correctness of programs.
Simple Semantics

<var-reference>
(<verb> <obj> ...)
(<special-word> <obj> ...)

Evaluation Rule

0. This rule is recursive.

1. If the expression is a <var-reference>, look up its value. If the variable is unbound, then the expression is an error.

2. Evaluate first item. If the first item is a <verb>, then evaluate the objects in some arbitrary order and apply the <verb> to the resulting object values. If the first item is a <special-word>, then use that word's evaluation rule. If the first item is neither a <verb> or a <special-word>, then the expression is an error.
Lexical Closures

\[
(define \text{foo} \\
(\text{let } ((x \ 2)) \\
(\lambda \ (y) (+ x y))))
\]

\[
(\text{foo} \ 2) \implies 4
\]

\[
(define \text{new-cons} \\
(\lambda \ (x \ y) \\
(\lambda \ (m) \\
(\text{if } (= m 0) \ x \\
(\text{if } (= m 1) \ y)))))
\]

\[
(\text{define \text{new-car}} \\
(\lambda \ (\text{cons}) \\
(\text{cons} \ 0)))
\]

\[
(\text{define \text{new-cdr}} \\
(\lambda \ (\text{cons}) \\
(\text{cons} \ 1)))
\]

\[
(\text{new-car} \ (\text{new-cons} \ 1 \ 2)) \implies 1 \\
(\text{new-cdr} \ (\text{new-cons} \ 1 \ 2)) \implies 2
\]
Data Abstraction

; Rational Number Construction
(define make-rat
  (lambda (n d) (cons n d)))

; Accessors
(define numer
  (lambda (x) (car x)))

(define denom
  (lambda (x) (cdr x)))

; Operations
(define +rat
  (lambda (x y)
    (make-rat
     (+ (* (numer x) (denom y))
        (* (denom x) (numer y)))
     (* (denom x) (denom y)))))

(define *rat
  (lambda (x y)
    (make-rat
     (* (numer x) (numer y))
     (* (denom x) (denom y))))
Functional Abstraction

(define flat-recursion
  (lambda (seed list-fn)
    (letrec
      ((helper
         (lambda (list)
           (if (null? list)
               seed
               (list-fn
                (car list)
                (helper (cdr list))))))
       helper)))

(define sum (flat-recursion 0 +))
(sum '(1 2 3 4 5)) ==> 15
(define product (flat-recursion 1 *))
(product '(1 2 3 4 5 6)) ==> 720
Functional Abstraction

(define deep-recursion
  (lambda (seed
    item-proc
    list-proc)
  (letrec
    ((helper
      (lambda (list)
        (if (null? list)
        seed
        (let
          ((a (car list)))
        (if (or
          (pair? a)
          (null? a))
          (list-proc
            (helper a)
            (helper (cdr list)))
          (item-proc
            a
            (helper (cdr
            list))))))))))

(helper))

(define product-all (deep-recursion 1 * *))
(product-all '((1 2) 3 (4 (5 6)))) ==> 720
(define flatten (deep-recursion '() cons append))
(flatten '((1 2) 3 (4 (5 6)))) ==> (1 2 3 4 5 6)
**Functional Programming**

It is possible to teach an entire introductory programming course using Scheme in a purely functional manner, i.e., without ever writing any code which modifies an already existant item. Such programs are free from side effects.

When used in this manner, Scheme provides a fundamentally different model of computation which is based on a different underlying model of a computer.

**Imperative Programming**

Scheme is not a pure functional language, so one can revert to a state changing model of computation complete with computational side effects, etc.

Scheme words which modify existant state use a "!" suffix in their names.

```
(define x 10)  ; bind x to a location which contains 10
(set! x +)  ; mutate the location bound to x to contain +

(define ls '(a b c))
(set-car! ls '(1 2))
ls ==> ((1 2) b c)
```
Recursion

Both recursion and iteration are expressed in a consistent manner through recursive procedures.

Recursive Processes

\[
\text{(define sum} \\
\quad (\text{lambda } (\text{ls}) \\
\quad \quad (\text{if } (\text{null? } \text{ls}) \\
\quad \quad \quad 0 \\
\quad \quad \quad (+ (\text{car } \text{ls}) (\text{sum } (\text{cdr } \text{ls})))))) \\
\text{(sum '(10 20 30)) }\Rightarrow 60
\]

Iterative Processes

\[
\text{(define sum1} \\
\quad (\text{lambda } (\text{ls}) \\
\quad \quad (\text{define sum-iter} \\
\quad \quad \quad (\text{lambda } (\text{ls acc}) \\
\quad \quad \quad \quad (\text{if } (\text{null? } \text{ls}) \\
\quad \quad \quad \quad \quad \text{acc} \\
\quad \quad \quad \quad \quad (\text{sum-iter} \\
\quad \quad \quad \quad \quad \quad (\text{cdr } \text{ls}) \\
\quad \quad \quad \quad \quad \quad (+ \text{acc } (\text{car } \text{ls})))))) \\
\quad \quad (\text{sum-iter } \text{ls } 0)) \\
\quad \quad (\text{sum1 '(10 20 30)) }\Rightarrow 60
\]
Continuations

The continuation of a subexpression, s, of an expression, e, is the computation in e which remains to be done after first evaluating s.

For example, suppose e is:

\((\star 2 (\, (+ 3 4)))\)

and s is:

\((+ 3 4)\)

We can write this continuation as a function of a single argument:

\((\text{lambda} \,(z) \, (\star 2 \, z))\)

So that the entire expression can be written as:

\(((\text{lambda} \,(z) \, (\star 2 \, z)) \, (+ 3 4))\)
Continuations

The continuation of the recursive call to sum in sum is:

\[(\text{lambda} \,(z) \,(+ \,(\text{car} \,ls) \,z))\]

The first continuation formed in \((\text{sum} \,'(10 \,20 \,30))\) is:

\[(\text{lambda} \,(z) \,(+ \,10 \,z))\]

The second is:

\[(\text{lambda} \,(z) \,(+ \,20 \,z))\]

The last:

\[(\text{lambda} \,(z) \,(+ \,30 \,z))\]

These three functions are composed, the last being applied to 0.

The recursive process is expressed as a functional composition.
Continuations

The continuation of the recursive call of sum1 in sum1 is:

\( \text{lambda} \ (z) \ z \)

the identity function. This procedure is tail-recursive. All Scheme interpreters and compilers optimize tail-recursion as an iterative loop.
Exact and Inexact Arithmetic

Scheme systems provide carefully designed systems for exact arithmetic as well as the usual inexact arithmetic found in conventional programming languages.

Parallel Processing

The Scheme evaluation rule for an application requires that the argument expressions be evaluated in arbitrary order. This means that side effects which occur in these expressions will be unpredictable and hence must be avoided. As a result, the underlying computational model allows the possibility of evaluating such expressions in parallel on parallel machines.
Object Programming

(define object-maker
  (lambda (init-value)
    (let ((object-value init-value))
      (lambda msg
        (case (car msg)
          ((value) object-value)
          ((set!) (set! object-value (cadr msg)))
          (else 'unknown-operation))))))

(define box (object-maker 6)) ==> <#unspecified>
box ==> <#procedure box>
(box 'value) ==> 6
(box 'set 7) ==> "undefined operation"
(box 'set! 7) ==> <#unspecified>
(box 'value) ==> 7
Object Programming

; An object oriented implementation of stacks

(define 1st car)
(define 2nd cadr)

(define invalid-method-name-indicator "unknown")

(define for-effect-only
  (lambda (item-ignored)
    "unspecified value")

(define delegate
  (lambda (obj msg)
    (apply obj msg)))

(define base-object
  (lambda msg
    (case (1st msg)
      ((type) "base-object")
      (else invalid-method-name-indicator)))
Object Programming

(define send
  (lambda args
    (let ((object (car args)) (message (cdr args)))
      (let ((try (apply object message)))
        (if (eq? invalid-method-name-indicator try)
            (error
              (string-append
                (symbol->string (car message)) "": "Bad method name sent to object of
                (object 'type) " type."
              try))))))))
Object Programming

\texttt{\textbf{define} stack-maker}
\texttt{lambda ()}
\texttt{(let ((stk '()))}
\texttt{(lambda msg}
\texttt{  (case (\texttt{1st} msg)}
\texttt{    ((\texttt{type}) "stack")}
\texttt{      ((\texttt{empty?}) (\texttt{null?} stk))}
\texttt{      ((\texttt{push!}) (\texttt{for-effect-only}}
\texttt{        (\texttt{set!} stk (\texttt{cons} (\texttt{2nd} msg) stk)))})
\texttt{      ((\texttt{top}) (\texttt{if} (\texttt{null?} stk))}
\texttt{        (\texttt{error} "top: The stack is empty.}}
\texttt{          (\texttt{car} stk))})
\texttt{      ((\texttt{pop!}) (\texttt{for-effect-only}}
\texttt{        (\texttt{if} (\texttt{null?} stk))}
\texttt{          (\texttt{error} "pop!: The stack is empty.
\texttt{            (\texttt{car} stk)))})
\texttt{      (\texttt{size}) (\texttt{length} stk))}
\texttt{    ((\texttt{print}) (display "TOP: ")}
\texttt{      (\texttt{for-each}}
\texttt{        (\texttt{lambda} (x)}
\texttt{          (display x)}
\texttt{          (display " "))}
\texttt{        stk)}
\texttt{      (newline))}
\texttt{    (\texttt{else} (\texttt{delegate base-object msg))))))))}
Macros

Scheme systems support a variety of powerful macro systems, including so-called *Hygenic Macros* which solve conflicts of different name spaces. The User usually has the choice of which type of macros to use from the Scheme standard library.
There is an effort to define a standard library for Scheme. The current library, slib2a2, available from the Scheme Repository ftp://ftp.cs.indiana.edu/pub/scheme-repository/code/lib/slib2a2.tar.gz contains the following components:

- "alist"
- "array"
- "array-for-each"
- "charplot"
- "collect"
- "common-list-functions"
- "debug"
- "defmacroexpand"
- "dynamic"
- "dynamic-wind"
- "fluid-let"
- "format"
- "format-inexact"
- "generic-write"
- "getopt"
- "hash"
- "hash-table"
- "line-i/o"
- "logical"
- "macro"
- "macro-by-example"
- "macros-that-work"
- "modular"
- "multiarg-apply"
- "multiarg/and-"
- "object"
- "object->string"
- "ol"
- "olsubs"
- "oop"
- "portable-scheme-debugger"
- "pprint-file"
- "pretty-print"
- "prime"
- "priority-queue"
- "process"
- "promise"
- "queue"
- "random"
- "random-inexact"
- "rationalize"
- "record"
- "record-object"
- "red-black-tree"
- "repl"
- "rev2-procedures"
- "rev3-procedures"
- "rev4-optional-procedures"
- "sort"
- "srgp"
- "stdio"
- "string-case"
- "string-port"
- "struct"
- "structure"
- "synchk"
- "syntactic-closures"
- "syntax-case"
- "test"
- "transcript"
- "tree"
- "useful"
- "values"
- "with-file"
- "x11"
- "xevent"
- "xm"
- "xmsubs"
- "xt"
- "xw"
- "yasos"
- "yasos-object"

To use functions in the sort library, use: (require 'sort)
Landmark Texts

Several important texts have been written which are suitable for use in introductory computer science courses.


Who is Using Scheme

Attached below is our most up-to-date compilation of colleges, universities, and secondary schools that are using Scheme in their curricula. Any additions, deletions, and/or corrections should be sent to Terry Kaufman at 71020.1774@compuserve.com.

Here are a few statistics regarding the schools using Scheme:

219 colleges/universities worldwide
  - 88 of these use Scheme in introductory courses
129 colleges/universities USA only
  - 47 of these use Scheme in introductory courses
34 secondary schools worldwide
28 secondary schools USA only
Many Free Implementations

The Scheme FAQ (Frequently Asked Questions), available from the Scheme Repository,

http://www.cs.cmu.edu:
8001/Web/Groups/AI/html/faqs/lang/scheme/top.html

The current Scheme FAQ lists 25 Scheme implementations which are available in source form at no charge on the Internet. One of the most widely used, Aubrey Jaffer's scm, has been ported to nearly any machine which might be in use today. scm is the version of Scheme used in GUILE, the new GNU extension language.

The FAQ also lists 4 commercial Scheme systems.