SCHEME: An Interpreter for Extended Lambda Calculus
Gerald J. Sussman and Guy L. Steele Jr.

Veit Heller
Papers We Love Berlin

February 16, 2020

- ▶ Introduction and historical context
- ▶ Scheme primer
- ▶ The good stuff
- ▶ Let's see some code!
- ▶ Implementation notes

Introduction

In 1975, a 21-year-old grad student named Guy Steele and his thesis advisor Gerald Sussman had something to show to the world: a little programming language called Scheme.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 349

December 1975

SCHEME

AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

by

Gerald Jay Sussman and Guy Lewis Steele Jr.

Figure: A wild paper appears.

The paper has all the goods a hacker could wish for: a reference, cool code examples, and an implementation of Lisp in Lisp.

The Name

The language was originally intended to be called SCHEMER, in reference to its ancestors PLANNER and CONNIVER.

Scheme: A primer

In Scheme, we define functions using define—you might know it as defn or defun in other Lisps:

```scheme
(define add
  (lambda (x y)
    (+ x y)))
```

Listing 1: Defining addition

NB: I eschewed the all-caps notation, and I hope your eyes will thank me for it.

We can quote things using either the function or the abbreviation '<thing>.

```
; this will always return the symbol x
(define gimme-x (lambda () 'x))
```

Listing 2: Using symbols as values

There is also the somewhat idiosyncratic `labels`, which allows you to define local functions that can be called inside a context, and can call themselves and other local functions in that context. You might know it as `letrec*` from later Schemes, and as simply `let` in Common Lisp.

```
; lets define cells!
(define cons-cell (lambda (contents)
    (labels ((the-cell
                (lambda (msg)
                  (if (eq msg 'contents) contents
                    (if (eq msg 'cell?) 'yes
                      (if (eq (car msg) '<-)
                        (block (aset 'contents (cadr msg))
                              the-cell)
                        (error '|Unrecognized Message - Cell|
                              msg
                              'wrng-type-arg)))))))
      the-cell)))
```
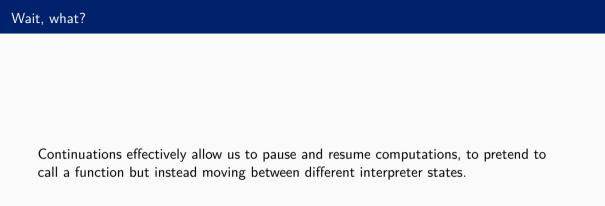
Listing 3: Let's define something!

There is more, though!

The good stuff

```
(define sqrt (lambda (x epsilon)
  ((lambda (ans looptag)
     (catch returntag ; setup return label
       (progn
         (aset 'looptag (catch m m)) ; setup loop label
         (if (< (abs (- (* ans ans) x)) epsilon)
             (returntag ans) ; goto return label
             nil) ; not done yet
         (aset 'ans (/ (+ (/ x ans) ans) 2.0)) ; calculate step
         (looptag looptag)))) ; goto loop label
   1.0
   nil)))
```

Listing 4: Jump around aka. "Sussman's favorite style/Steele's least favorite"

Continuations effectively allow us to pause and resume computations, to pretend to call a function but instead moving between different interpreter states.

Wait, what?

It's pretty mind-bending at first and I understand if it's a little much.

The paper is a bit harsh here: "Anyone who doesn't understand how this manages to work probably should not attempt to use CATCH."

As if that wasn't enough, we also have a multiprocessing story. We can create new processes using `create!process`, start them using `start!process`, stop them using `stop!process`, and synchronize using `evaluate!uninterruptibly`.

"This concludes the SCHEME Reference Manual."

Code Examples

```
(define fact (lambda (number continuation)
  (if (= number 0)
    (continuation 1)
    (fact (- number 1)
          (lambda (a) (continuation (* number a)))))))

; simple computation
(fact 5 (lambda (x) x))

; computation, but we log each step
(fact 5 (lambda (x) (block (print x) x)))
```

Listing 5: Factorial, but the computation happens in continuations.

Consider you have two functions, and you want to run them in parallel, stopping whenever the first one terminates, and returning its result.

The authors call this "A Useless Multiprocessing Example".

```
(define try!two!things!in!parallel (lambda (f1 f2)
  (catch c
    ((lambda (p1 p2)
      ((lambda (f1 f2)
        ; ensures both fs get started atomically
        (evaluate!uninterruptibly
          (block (aset 'p1 (create!process '(f1)))
                 (aset 'p2 (create!process '(f2)))
                 (start!process p1)
                 (start!process p2)
                 (stop!process **process**)))) ; stop yourself
        ; what are f1 and f2?
        ))
      nil nil)))))
```

Listing 6: Dont!Shout!At!Me

```
; f1 =
(lambda ()
  ; stop the other process and return
  ((lambda (value)
    (evaluate!uninterruptibly
      (block (stop!process p2) (c value))))
   (f1))) ; do our thing
; f2 =
(lambda ()
  ((lambda (value)
    (evaluate!uninterruptibly
      (block (stop!process p1) (c value))))
   (f2)))
```

Listing 7: The magic bits

Let's consider a simple pattern matching function.

```
; ! = zero or more things (.* in regex)
; ? = any single thing (. in regex)
; anything else = itself

(match '(A !B ?C ?C !B !E)
       '(A X Y Q Q X Y Z Z X Y Q Q X Y R))
```

Listing 8: A simple pattern matcher.

Instead of just returning the match groups, however, we return the match groups and a continuation that gives us backtracking and will return alternative matches, Prolog-style.

How would we implement this?

Pattern matching!

Figure: A simple solution to a simple problem.

If you have the time to study the code examples, keep your eyes peeled for the definition of the `do` macro and the `samefringe` problem.

Implementation Notes

TODO: Talk about the implementation.