SCHEME: An Interpreter for Extended Lambda Calculus

Gerald J. Sussman and Guy L. Steele Jr.

Veit Heller
Papers We Love Berlin

February 24, 2020

Agenda

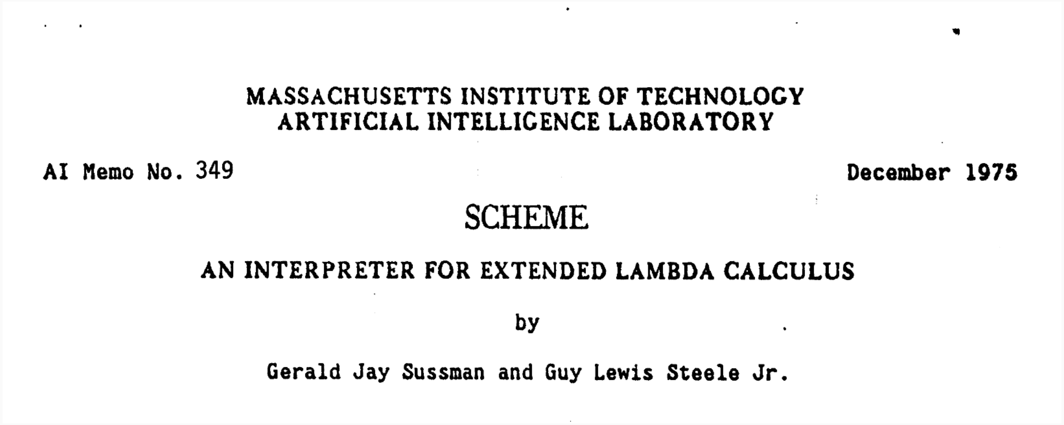
- Introduction and historical context
- Scheme primer
- The good stuff
- Let's see some code!
- Implementation notes

Introduction

The Paper

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.

The Paper



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

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

Scheme: A primer

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

```
(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.

Scheme: A primer

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

Scheme: A primer

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.

Putting it all together

```
; 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!

And now?

There is more, though!

The good stuff

Continuations!

```
(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”

Wait, what?

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

Wait, what?

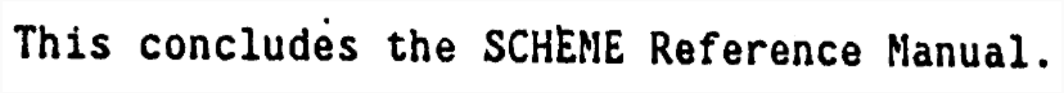
It’s pretty mind-bending at first and I understand if it’s a little much.

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

Multiprocessing

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`.

Oof.



This concludes the SCHEME Reference Manual.

Code Examples

Continuations, yet again!

```
(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.

Multiprocessing

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”.

Multiprocessing

```
(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

Multiprocessing

```
; 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

Pattern matching!

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))
```

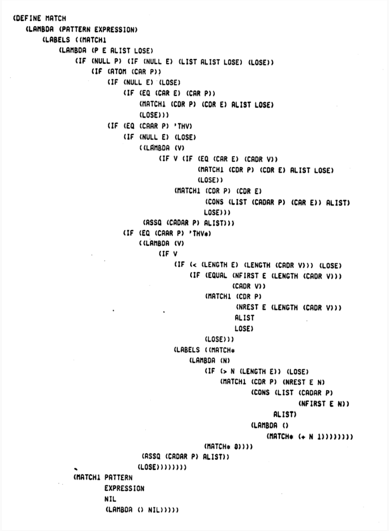
Listing 8: A simple pattern matcher.

Pattern matching!

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!



```
(DEFINE MATCH
   (LAMBDA (PATTERN EXPRESSION)
       (LABELS ((MATCH1
           (LAMBDA (P E ALIST LOSE)
               (IF (NULL P) (IF (NULL E) (LIST ALIST LOSE) (LOSE))
                   (IF (ATOM (CAR P))
                       (IF (NULL E) (LOSE)
                           (IF (EQ (CAR E) (CAR P))
                               (MATCH1 (CDR P) (CDR E) ALIST LOSE)
                               (LOSE)))
                       (IF (EQ (CAAR P) 'THV)
                           (IF (NULL E) (LOSE)
                               ((LAMBDA (V)
                                   (IF V (IF (EQ (CAR E) (CADR V))
                                             (MATCH1 (CDR P) (CDR E) ALIST LOSE)
                                             (LOSE))
                                       (MATCH1 (CDR P) (CDR E)
                                               (CONS (LIST (CADAR P) (CAR E)) ALIST)
                                               LOSE)))
                                (ASSQ (CADAR P) ALIST)))
                           (IF (EQ (CAAR P) 'THV*)
                               ((LAMBDA (V)
                                   (IF V
                                       (IF (< (LENGTH E) (LENGTH (CADR V))) (LOSE)
                                           (IF (EQUAL (NFIRST E (LENGTH (CADR V)))
                                                      (CADR V))
                                               (MATCH1 (CDR P)
                                                       (NREST E (LENGTH (CADR V)))
                                                       ALIST
                                                       LOSE)
                                               (LOSE)))
                                       (LABELS ((MATCH*
                                           (LAMBDA (N)
                                               (IF (> N (LENGTH E)) (LOSE)
                                                   (MATCH1 (CDR P) (NREST E N)
                                                           (CONS (LIST (CADAR P)
                                                                       (NFIRST E N))
                                                                 ALIST)
                                                           (LAMBDA ()
                                                               (MATCH* (+ N 1))))))))
                                               (MATCH* 0))))
                                (ASSQ (CADAR P) ALIST))
                               (LOSE))))))))
               (MATCH1 PATTERN
                      EXPRESSION
                      NIL
                      (LAMBDA () NIL)))))
```

Figure: A simple solution to a simple problem.

Examples I wish I would have had time for

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

The interpreter

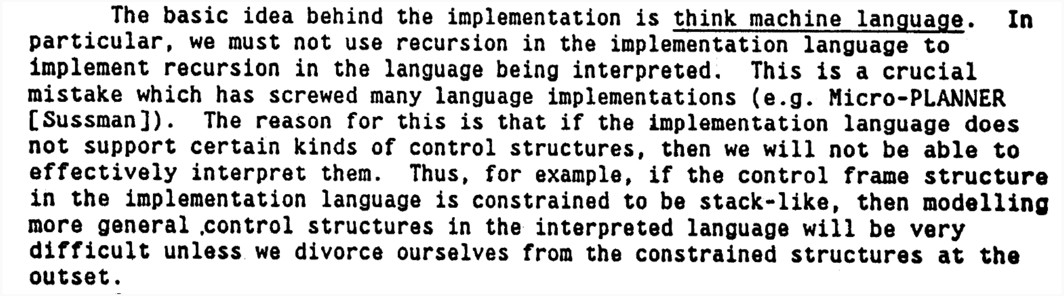
The design of the interpreter is deceivingly straightforward.

The interpreter

It is

- lexically scoped,
- based on continuations and “clinks”,
- built on top of MacLISP (all of MacLISP is available as a primitive),
- not purely functional, and
- concurrent (but not parallel).

The implementation



The basic idea behind the implementation is think machine language. In particular, we must not use recursion in the implementation language to implement recursion in the language being interpreted. This is a crucial mistake which has screwed many language implementations (e.g. Micro-PLANNER [Sussman]). The reason for this is that if the implementation language does not support certain kinds of control structures, then we will not be able to effectively interpret them. Thus, for example, if the control frame structure in the implementation language is constrained to be stack-like, then modelling more general control structures in the interpreted language will be very difficult unless we divorce ourselves from the constrained structures at the outset.

The implementation

It is striking to me how readable the implementation is.

The implementation

If you tried to read the implementation but gave up, here are a few pointers:

- Start in the evaluation loop.
- Find out how components work.
    - "Where do clinks pass through?"
    - "What happens when I call `catch`?"
- Trace a program (if you must).

The implementation

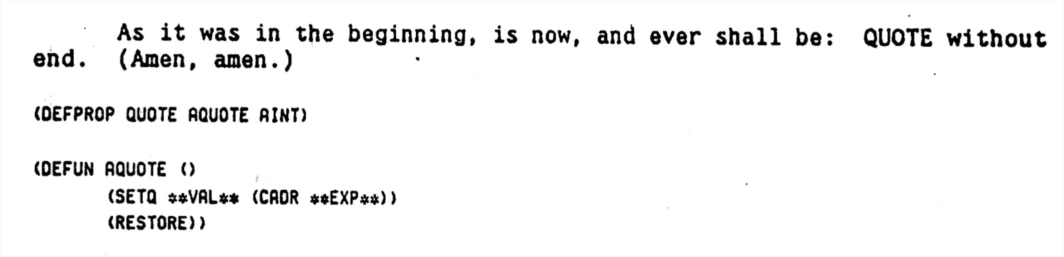
Let's look at a primitive: `if`.

```
(defun if ()
  (saveup 'if1) ; we will go there next
  (setq **exp** (cadr **exp**) ; first eval the condition
        **pc** 'aeval))

(defun if1 ()
  (cond
    (**val** ; the condition was true, take then
      (setq **exp** (caddr **exp**))
     t ; otherwise take else
      (setq **exp** (cadddr **exp**))))
  (setq **pc** 'aeval))
```

Listing 9: The internals of `if`.

Amen



As it was in the beginning, is now, and ever shall be: QUOTE without end. (Amen, amen.)

```
(DEFPROP QUOTE AQUOTE AINT)

(DEFUN AQUOTE ()
       (SETQ **VAL** (CADR **EXP**))
       (RESTORE))
```

That's it?

Cool, so that's all there is to it?

That’s it?

Let’s talk about the multiprocess primitives really quick.

Implementing threads

This is what we need to do:

- Creating a process creates new registers and sets up a new interpreter
- Starting a process puts it in the process queue.
- Stopping a process removes it from the process queue and terminates it if it is the current process.
- Evaluating uninterruptibly means binding a flag to `nil` that tells the interpret er not to swap processes.
- Then we need to build a magical scheduler that swaps process in the queue once in a while.

Ta-dah!

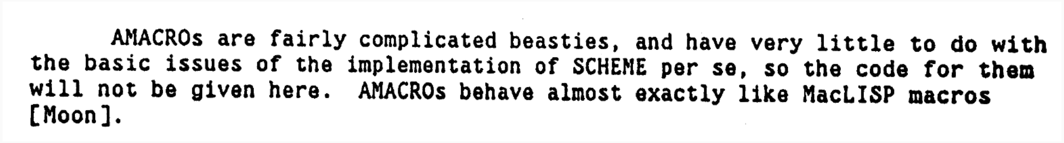
Implementing continuations

What about continuations?

- `catch` tells the evaluator to switch the clink.
- Then we need to know how and when to switch these out in the main loop (hint: look at `evlis`).
- Swapping process becomes almost equivalent to swapping continuations!

Ta-dah!

Implementing macros


AMACROs are fairly complicated beasties, and have very little to do with the basic issues of the implementation of SCHEME per se, so the code for them will not be given here. AMACROs behave almost exactly like MacLISP macros [Moon].

Fin

This is the end of the SCHEME interpreter!

References

- Gerald Jay Sussman and Guy L. Steele, Jr.: Scheme—An Interpreter for Extended Lambda Calculus
- Guy L. Steele, Jr.: Rabbit—A Compiler for Scheme
- These slides: `https://github.com/hellerve/talks`
- A series of blog posts on Scheme macros: `https://blog.veitheller.de/scheme-macros`

The End

Thank you!

Questions?

Slides at `https://github.com/hellerve/talks`