



The Swine Before Perl

Shriram Krishnamurthi
Brown University and PLT



Why We're Here

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

—Phil Greenspun's
Tenth Law of Programming

Our Corollary

Any sufficiently useful C or Fortran program
needs to be “sufficiently complicated”

We’re here to provide the heavy lifting

I’m here to motivate how Scheme does this



The Swine: PLT Scheme

“Whatever: can you
program in it?”

Books

- *Teach Yourself Scheme in Fixnum Days*
 - *How to Use Scheme*
 - Extensive manuals and user support
 - **It's all free of cost**
-

Software #1: MzScheme

- Quick, textual shell: ideal for scripting
 - Rich module system
 - Rich mixin-based object system
 - Embeddable in applications
 - Garbage collection across C/C++/Scheme
 - Libraries for lots of 3- and 4-letter acronyms (XML, CGI, ODBC, COM, ...)
-

MzScheme

Welcome to MzScheme version 103, Copyright (c) 1995-2000 PLT (Matthew Flatt)

> <require-library "function.ss">

> <require-library "url.ss" "net">

>

<foldl + 0 <map file-size <directory-list>>>

3887958

>

<call/input-uri <string>url "http://www.cs.brown.edu/~sk/test.html">
get-pure-port display-pure-port)

<html>

<head><title>Test</title></head>

<body><p>Hello!</p></body>

</html>

>

Software #2: DrScheme

- Fully-graphical programming environment
 - Full portability: Unix, Mac and Windows
 - Special support for beginning Schemers
-

auto-talk-2.ss - DrScheme

File Edit Windows Show Language Scheme Help

auto-talk-2.ss Save Analyze Check Syntax Step Execute Break

```
(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))]
                              (let ([1st (first stream)])
                                (let ([new-state (assv 1st transitions)])
                                  (if new-state
                                      (walker (cadr new-state) (rest stream))
                                      false)))))))]))
  (walker 'init stream)))
```

Welcome to [DrScheme](#), version 103p1.
Language: [Graphical Full Scheme \(MrEd\)](#).

```
#t
#f
> (b-machine '(c a d d a d r))
#t
> |
```

7:3 Unlocked not running

Software #3: Type Inference

- Two PhD theses and counting
 - Graphically displays inferred types
 - Let's see how it works ...
-

```
auto-talk-2.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help

(define b-machine-states
  '( (init (c loop))
      (loop (a loop)
            (d loop)
            (r end))
      (end (r end))))

(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))])
                          (let ([1st (first (car stream))]
                                (let ([new-state (assv 1st transitions)])
                                  (if new-state
                                      (walker (cadr new-state) (rest stream))
                                      false)))))))]))
    (walker 'init stream)))

(b-machine '(c a d a d r))
(b-machine '(c a d a r d))

Welcome to MrSpidey, version 103p1.
CHECKS:
car check in file "auto-talk-2.ss": line 1, column 15
cdr check in file "auto-talk-2.ss": line 18, column 34
car check in file "auto-talk-2.ss": line 19, column 47

Unlocked 
```

```
auto-talk-2.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help

(define b-machine-states
  '( (init (c loop))
      (loop (a loop)
            (d loop)
            (r end))
      (end (r end))))

(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))])
                          (let ([lst (first (car stream (listof sym)))]))
                            (let ([new-state (assv lst transitions)])
                              (if new-state
                                  (walker (cadr new-state) (rest stream))
                                  false)))))))]])
    (walker 'init stream))

(b-machine '(c a d a d r))
(b-machine '(c a d a r d))

Welcome to MrSpidey, version 103p1.
CHECKS:
car check in file "auto-talk-2.ss": line 1, column 15
cdr check in file "auto-talk-2.ss": line 18, column 34
car check in file "auto-talk-2.ss": line 19, column 47

Unlocked
```

```
auto-talk-2.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help

(define b-machine-states
  '( (init (c loop))
      (loop (a loop)
            (d loop)
            (r end))
      (end (r end))))

(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))]
                              (let ([lst (first (car stream) (listof sym))]
                                  (let ([new-state (assv lst transitions)]
                                      (if new-state
                                          (walker (cadr new-state) (rest stream))
                                          false)))))))]))

    (walker 'init stream)))

(b-machine '(c a d a d r))
(b-machine '(c a d a r d))

Welcome to MrSpidey, version 103p1.
CHECKS:
car check in file "auto-talk-2.ss": line 1, column 15
cdr check in file "auto-talk-2.ss": line 18, column 34
car check in file "auto-talk-2.ss": line 19, column 47

Unlocked
```

```
auto-talk-2.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help

(define b-machine-states
  '( (init (c loop))
      (loop (a loop)
            (d loop)
            (r end))
      (end (r end))))

(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))]
                            (let ([lst (first (car stream) (listof sym))]
                                (let ([new-state (assv lst transitions)]
                                    (if new-state
                                        (walker (cadr new-state) (rest stream))
                                        false)))))))]))
          (walker 'init stream)))

(b-machine (c a d a d r))
(b-machine (c a d a r d))

Welcome to MrSpidey, version 103p1.
CHECKS:
car check in file "auto-talk-2.ss": line 1, column 15
cdr check in file "auto-talk-2.ss": line 18, column 34
car check in file "auto-talk-2.ss": line 19, column 47

Unlocked
```

```
auto-talk-2.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help

(define b-machine-states
  '( (init (c loop))
      (loop (a loop)
            (d loop)
            (r end))
      (end (r end))))

(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))]
                              (let ([lst (first (car stream) (listof sym) )])
                                (let ([new-state (assv lst transitions)])
                                  (if new-state
                                      (walker (cadr new-state) (rest stream))
                                      false)))))))]))
    (walker 'init stream)))

(b-machine '(c a d a d r))
(b-machine '(c a d a r d))

Welcome to MrSpidey, version 103p1.
CHECKS:
car check in file "auto-talk-2.ss": line 1, column 15
cdr check in file "auto-talk-2.ss": line 18, column 34
car check in file "auto-talk-2.ss": line 19, column 47

Unlocked
```

```
auto-talk-2.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help

(define b-machine-states
  '( (init (c loop))
      (loop (a loop)
            (d loop)
            (r end))
      (end (r end))))

(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))]
                            (let ([lst (first (car stream) (listof sym))]
                                (let ([new-state (assv lst transitions)]
                                    (if new-state
                                        (walker (cadr new-state) (rest stream) (listof sym))
                                        false)))))))]))

  (walker 'init stream)))

(b-machine '(c a d a d r))
(b-machine '(c a d a r d))

Welcome to MrSpidey, version 103p1.
CHECKS:
car check in file "auto-talk-2.ss": line 1, column 15
cdr check in file "auto-talk-2.ss": line 18, column 34
car check in file "auto-talk-2.ss": line 19, column 47

Unlocked
```


Software #4: Web Server

- Dynamic content generation is a breeze
 - HTML/XML transformation is a breeze
 - Trivially build use-once security policies
 - For dynamic content, 8x speed of Apache
-

The Gems

- Closures
- Continuations

The Crown Jewels

- That stupid parenthetical syntax
- Macros
- Tail calls

The
Lesser
Gems



A Pearl

On Stealing Beauty –
Where you should start

Problem

Pattern-matcher for streams:

- Consumes a stream of input tokens
 - Must be easy to write and read
 - Must be fairly fast
 - Must integrate well into rest of code
-

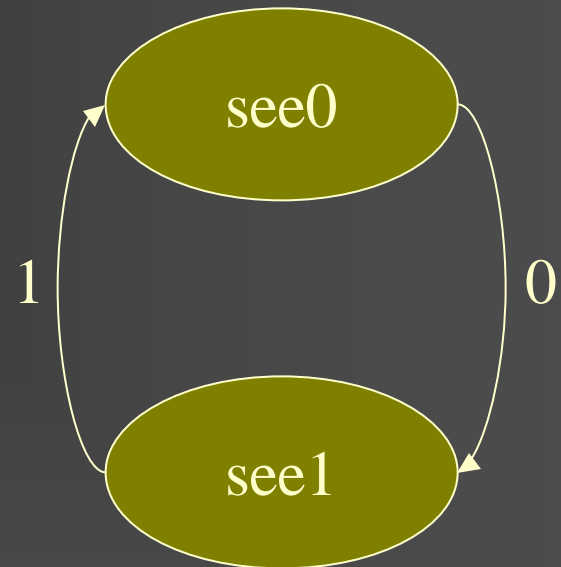
Let's Think About Automata

I want to be able to write

```
automaton see0
```

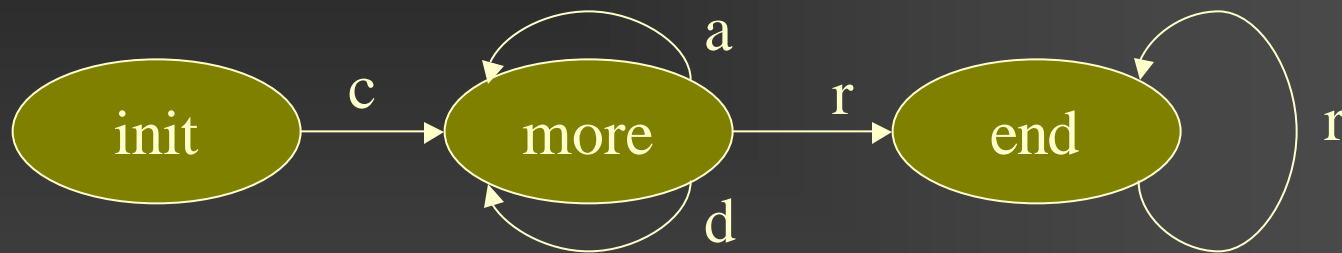
```
see0 : 0 → see1
```

```
see1 : 1 → see0
```



Another Example

car, cdr, cadr, caddr, cdar, caddr, ...



init : c → more

more : a → more

d → more

end : r → end

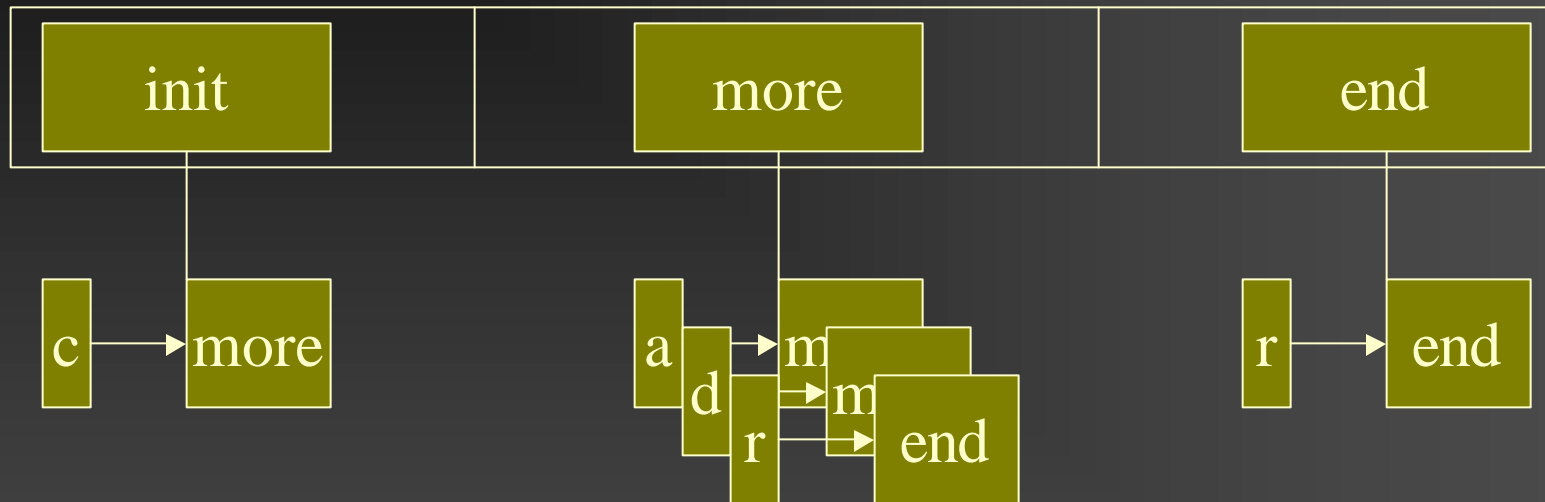
r → end

Let's Look at That Again

```
automaton init
  init : c → more
  more : a → more
        d → more
        r → end
  end   : r → end
```

How would **you** implement it?

Natural Solution



If stream ends, accept

If no next state found, reject

If next state found, continue

First Version

- Development time: 12 minutes
 - Bugs: 2
 - Performance (PLT Scheme):
 - 10000 elements: 50 ms
 - 100000 elements: 440 ms
 - 1000000 elements: 4316 ms
-

The Code

```
(define b-machine-states
  '((init (c more))
    (more (a more)
          (d more)
          (r end))
    (end (r end))))

(define (b-machine stream)
  (letrec ([walker (lambda (state stream)
                    (or (empty? stream)
                        (let ([transitions
                              (cdr (assv state b-machine-states))])
                          (let ([1st (first stream)])
                            (let ([new-state (assv 1st transitions)])
                              (if new-state
                                  (walker (cadr new-state) (rest stream))
                                  false)))))))]
    (walker 'init stream)))
```

What's The Essence?

- Per state, fast conditional dispatch table
 - An array of states
 - Quick state transition
-

A Message From Our Sponsors

We will encourage you to develop the three great virtues of a programmer: laziness, impatience, and hubris.

—Larry Wall and Randal L Schwartz

Thinking Virtuously

- Per state, fast conditional dispatch table
Compiler writers call this “case ... switch”
 - An array of states
Function pointers offer random access
 - Quick state transition
If only function calls were implemented as “goto”s ...
-

In Other Words: `init` State Would Become

`init` \equiv

```
(procedure (stream)
```

```
  (or (empty? stream)
```

```
    (case (first stream)
```

```
      [c (more (rest stream))]
```

```
      [else false])))
```



more


In Other Words: more State Would Become

```
more ≡  
(procedure (stream)  
  (or (empty? stream)  
    (case (first stream)  
      [a (more (rest stream))]  
      [d (more (rest stream))]  
      [r (end (rest stream))]  
      [else false])))) → end
```

In Other Words: The Whole Code Would Become

```
(define b
  (letrec ([init
            (procedure (stream)
              (or (empty? stream)
                  (case (first stream)
                    [c (more (rest stream))]
                    [else false]))))]
            [more
            (procedure (stream)
              (or (empty? stream)
                  (case (first stream)
                    [a (more (rest stream))]
                    [d (more (rest stream))]
                    [r (end (rest stream))]
                    [else false]))))]
            [end
            (procedure (stream)
              (or (empty? stream)
                  (case (first stream)
                    [r (end (rest stream))]
                    [else false]))))]
            init))
```


Scoreboard

- Laziness: 
 - Impatience: nope; too much code
 - Hubris: ...
-

In General

```
(state : (label → target) ...)
```

→

```
(procedure (stream)
  (or (empty? stream)
    (case (first stream)
      [label (target (rest stream))]
      ...
      [else false]))))
```

Even More Generally

```
(_ init-state  
  (state : (cndn -> new-state) ...)  
  ...)
```



```
(letrec ([state  
  (procedure (stream)  
    (or (empty? stream)  
        (case (first stream)  
          [cndn (new-state (rest stream))]  
          ...  
          [else false]))))]  
  ...)  
  init-state)
```

In Fact, That's the Code!

```
(define-syntax automaton
  (syntax-rules (-> :)
    [[input pattern
      output pattern] ] ) )
```

This is a Scheme macro

The Automaton

automaton init

init : c → more

more : a → more

d → more

r → end

end : r → end

In Scheme

```
(automaton init
  (init : (c → more))
  (more : (a → more)
         (d → more)
         (r → end))
  (end  : (r → end)))
```

What a Schemer really sees

```
(automaton init
  (init : (c → more))
  (more : (a → more)
         (d → more)
         (r → end))
  (end  : (r → end)))
```


With Clients

```
(define (v s)
  ((if (eq? (first s) 'c)
       (automaton init
        (init : (c -> loop))
        (loop : (a -> loop)
              (d -> loop)
              (r -> end))
        (end : (r -> end)))
       (automaton see0
        (see0 : (0 -> see1))
        (see1 : (1 -> see0))))
    s))
```


Second Version

- Development time: 5 minutes
 - Bugs: 0
 - Performance:
 - 10000 elements: 30 ms
 - 100000 elements: 310 ms
 - 1000000 elements: 3110 ms
-

Scoreboard

- Laziness: 
 - Impatience: 
 - Hubris: stay tuned
-

What Really Happened

The traditional implementation is an
Interpreter

The macro system implements a
Compiler

from Scheme++ to Scheme – and lets you
reuse the existing Scheme compiler

Macros

- Clean, convenient spec of automata
 - Permits *nested* ... – “pattern matching”
 - Easy to create domain-specific language
 - Each module can have different macros
-

Tail Calls

Ensures that

state transition =

goto =

loop for free!

Notice tail *recursion* isn't enough!

(Oh, and try generating loop code ...)

Stupid Parenthetical Syntax

```
(automaton see0  
  (see0 (0 -> see1))  
  (see1 (1 -> see0)))
```

is clearly ugly, evil, and an insidious plot
hatched by misbegotten academics

Smart Parenthetical Syntax

```
<automaton see0
  <state name="see0">
    <trn> <from> 0 </from>
      <to> see1 </to> </trn> </state>
  <state name="see1">
    <trn> <from> 1 </from>
      <to> see0 </to> </trn> </state>
</automaton>
```

is a hip, cool, great new idea

Python vs. Scheme (python.org)

- Standard object system
 - Regular expressions, Internet connectivity
 - Many builtin data types
 - One standard implementation
 - Relatively main-stream syntax
 - Main-stream control structures
-

Python vs. PLT Scheme (python.org)

- ✓ Standard object system
 - ✓ Regular expressions, Internet connectivity
 - ✓ Many builtin data types
 - ✓ One standard implementation
 - Relatively main-stream syntax – at what price?
 - Main-stream control structures
 - We got macros – you got five minutes?
 - Real Programmers use map/filter/fold, tail calls, ...
-

Take-Home Morals

- If you claim to be smart, be *really* smart about reuse
 - Scheme fits together particularly cleverly – you won't get it just by reading about it, you'll only *think* you did
 - People who don't understand this use of tail calls don't get it
 - Take a real languages course in college
-

Scoreboard

■ Laziness: 

■ Impatience: 

■ Hubris: 

A Parting Thought

- A REPL is a Read-Eval-Print Loop:
Read, then Evaluate, then Print, then Loop
 - In code:
`Print (Eval (Read ()))`; Loop
 - A Print-Eval-Read Loop
 - A Print-Eval-Read Loop
-

Obligatory URL

<http://www.plt-scheme.org/>

(Thanks: Matthias, Matthew, Robby, John, Paul, Paul, Jamie, Philippe, Dorai, and dozens others)
