Perl in Scheme: A DSL

Abram Hindle

Kitchener/Waterloo Perl Mongers
Canada
http://kw.pm.org/

{abez+kw}@abez.ca

Overview

- Motivation
- Scheme Intro
- DSL
- Conclusions

Introduction

- I like Scheme
- Scheme is a tiny but powerful language
- Scheme's liberal syntax and macro language allows it to be very flexible
- Scheme has a different style for implementing tasks
- Sometimes I have solved the problem already in Perl
  - But I'm working in Scheme!
    - Sometimes vice versa

DSL

- Domain Specific Language
- I’m claiming the macros and functions in this presentation form a DSL because they provide new syntax
- It’s a really weak DSL so far
- Perl in Scheme

Scheme

- Scheme is a lisp derived programming language
- Lexically scoped (static scope)
- Supports Functional programming
- Minimalist
- Continuations
- Tried to be similar to lambda calculus

Scheme

- Types and syntax
  - Numbers (integers, floats, rationals, etc.)
  - Symbols ’this-is-a-symbol
  - procedures (f x)
  - Strings “Hi!”
  - chars #

    A (Capital A)
  - pair (a . b)

- list (a b c)
- booleans #t #f
- Vector
- ports

Scheme

- Uses s-expressions
  - Prefix notation, not infix.
  - (function-name param1 param2 param3)
  - (= (+ (* 1 2) 3) 6)

Scheme

- cons
  - Just about everything in scheme is made up of linked lists
  - (cons a b) makes a cons cell of a b
    - (a . b)
    - (list 1 2 3) versus (cons 1 (cons 2 (cons 3)))

Scheme

- car
  - car gets the head of a cons
    - (car (list 1 2 3)) is 1
    - (car (cons 1 ’())) is 1

Scheme

- cdr
  - cdr gets the tail of a cons
    - (cdr (cons 1 2)) is 2
    - (cdr (cons 1 (cons 2 ’()))) is (2) or (cons 2 ’())

Scheme

- Composition of car and cdr
  - cadr - gets the head of the tail of a cons cell
    - (cadr (list 1 (list 2))) is 2
    - (cadr ’(1 (2))) is 2

Scheme

- Procedures
  - lambda produces an anonymous function
  - define produces a named function
  - lambda can be assigned in defines and lets

Scheme

- Procedures
  - (lambda (x) x) - identity of 1 argument
  - (define (identity x) x) - named function identity
  - (define (square x) (* x x))

Scheme

- Macros
  - Functions which return lists of symbols which look like scheme code that become scheme code
  - Get around problems that simple function calls cannot

Chicken

- We’ll be using chicken scheme
- By Felix L. Winkelmann
- http://callcc.org/

Making Scheme Useful

- Scheme is tiny
  - R5RS is less than 60 pages

```
(require—extension posix)
(require—extension srfi—1) ;; lists @_@
(require—extension srfi—13) ;; strings @_@
```

Slurpfile

-
  ```
  (define (slurp-file file)
          (with-input-from-file file read-all-lines))
  ```

Perl ENV

-
  ```
  (define (ENV x) (cdr (assoc x (current—environment)))))
  ```

Perl pop

- Note we’re using a macro

```
(define—macro pop!
    (lambda (x)
        ‘(if (null? ,x) #f
                (let ((r (reverse ,x)))
                    (let ((k (car r)))
                        (set! ,x (reverse (cdr r)))
                        k)))))
```

Perl push

- Note we’re using a macro

```
(define-macro push!
        (lambda (x y) `(set! ,x (reverse (cons ,y (reverse ,x))))))
```

Perl unshift!

- Note we’re using a macro

```
(define-macro unshift!
        (lambda (x y) `(set! ,x (cons ,y ,x))))
```

Perl shift!

- Note we’re using a macro

```
(define—macro shift!
        (lambda (x)
                `(if (null? ,x)
                                #f
                                (let ((k (car ,x)))
                                                (set! ,x (cdr ,x)) k))))
```

Perl ARGV

-
  ```
  (define (perl—argv)
          (let ((args (argv)))
                  (if (and (pair? args) (string=? ”csi” (pathname—file (car args))))
                          (cddr args)
                          (cdr args))))
  ```

Perl my

-
  ```
  (define—macro begin—my
          (let ((ismy? (lambda (x) (and (pair? x) (eqv? 'my (car x))))))
                  (lambda x
                           (let loop ((l x))
           (if (pair? l)
                           (let ((head (car l)))
                                   (if (ismy? head)
                           (list 'let
                           (list (list (cadr head) (caddr head)))
                           (loop (cdr l)))
                           (if (pair? (cdr l))
                                           (list 'begin (car l) (loop (cdr l)))
                                           (list 'begin (car l))))))))))
  ```

Perl my

- Here’s a test case

```
(define (begin—my—test)
          (begin—my
           (my a 99)
           (my b 2)
           (set! a (+ a b)) ;103
           (print a)
           (set! a (+ a b)) ;105
           (print a)
           (my c 3)
           (my a (+ a c))
           (print a)
           a))
```

But what about hashes!

- Most schemes have a hash implementation, R6RS has one
  - R5RS doesn't
- SRFI-69
- The hash functions are very wordy
- We'll need this code
- Perl's auto vivification can cause us problems

```
(define (hash—cons—key hash—table key value)
(hash-table-update!/default hash-table key (lambda (x) (cons value x)) '())
```

A single hashtable

-
  ```
  (define (make-easy-hash)
          (let* ((h (make-hash-table))
                  (acc
                          (lambda (x . y)
                                  (case x
                                          ('get (hash-table-ref h (car y)))
                                          ('set (hash-table-set! h (car y) (cadr y)))
                                          ('has (hash-table-exists? h (car y)))
                                          ('del (hash-table-delete! h (car y)))
                                          ('cons (hash-cons-key h (car y) (cadr y)))
                                          ('alist (hash-table->alist h))
                                          (else (error (list "Don't␣know␣what␣this␣is:␣" x)))))))
                  acc))
  ```

A single hashtable

- We can use it like so:

```
(let ((h (make—easy—hash)))
        (h ’set ”lol” ”hy”)
        (print (h ’has ”lol”))
        (print (h ’get ”lol”))
        (h ’del ”lol”)
        (print (h ’has ”lol”)))
```

What about nested hash tables?

- Well lets just hack it and pretend we did it

```
(define (make-easy-n-key-hash)
        (define (mk-key keys) keys)
        (define (get-key y) (mk-key (car y)))
        (define (get-arg y) (cadr y))
        (define (get-arg-no-key y) (car y))
        (let* ((h (make-hash-table equal?))
                 (acc
                       (lambda (x . y)
                               (case x
                                       ('get (hash-table-ref h (get-key y)))
                                       ('set (hash-table-set! h (get-key y) (get-arg y)))
                                       ('has (hash-table-exists? h (get-key y)))
```

```
                                                          ('del (hash—table—delete! h (get—key y)))
                                                          ('cons (hash—cons—key h (get—key y) (get—arg y)))
                                                          ('alist (hash—table—>alist h))
                                                          ('get—nth—keys (delete—duplicates!

                                                          (else (error (list "Don't␣know␣what␣this␣is:␣" x)))))))
                                  acc))
```

What about nested hash tables?

- How do we use them?

```
(define (test—easy—n—hash)
        (define h (make—easy—n—key—hash))
        (h ’set ’(”abram” ”loves”) ”lixin”)
        (h ’set ’(”abram” ”hates”) ”work”)
        (h ’set ’(”lixin” ”loves”) ”abram”)
        (list
         (h ’get ’(”abram” ”loves”))
         (h ’get ’(”lixin” ”loves”))
         (h ’get ’(”abram” ”hates”))))

(test—easy—n—hash)
```

Convienance

```
(define eq string=?)
(define ne (lambda (x y) (not (eq x y))))
(define eq string=?)
(define ne (lambda (x y) (not (eq x y))))
(define (subst from to str)
        (string—substitute from to str))
(define unlink delete—file*)
(define link file—link)
```

Perl while(<>)

-
  ```
  (define (read—lines—from—<> filelist f)
      (define (handle—file file)
          (lambda (filename)
              (with—input—from—file filename
                  (lambda ()
                      (handle—each—line f)))))
      (define (handle—stdin) (handle—each—line f))
      (cond
       ((pair? filelist)      (for—each handle—file filelist))
       ((or (and (string? filelist) (eq filelist "—"))
            (list? filelist))
          (handle—stdin))
       ((and (string? filelist)) (handle—file filelist))
       (else (error (list "What␣is␣" filelist) ))))
  ```

Conclusions

- Tiny DSL of perl
  - Taking some perl features and reimplementing them
  - Making functions so it is easy to follow the perl way
  - Barely scratched the surface
  - sometimes it is useful to program the perl way

Future Work

- Integrate Regular Expressions better
  - Chicken has regular expressions but there are add-ons for good syntax
- maybe a perl-¿scheme convertor, or a true perl DSL
- Handle symbols properly, handle namespaces
- HEREDOC syntax?
- Maybe more useful would be actually loading up perl and making an interface such that perl libraries could

be called.

Reading

- http://practical-scheme.net/wiliki/schemexref.cgi
- http://www.schemers.org/Documents/Standards/R5RS/HTML/
- http://srfi.schemers.org/srfi-1/srfi-1.html
- http://srfi.schemers.org/srfi-13/srfi-13.html
- http://callcc.org
- http://chicken.wiki.br/Unit%20regex

Thank you

- Any Questions?