iSAQB Advanced DSL - DSLs as Libraries

Michael Sperber

Created: 2024-06-02 Sun 17:04

(LG 5-3) Macros in Racket

```
(define-syntax-rule (swap! x y)
  (let ((z x))
    (set! x y)
    (set! y z)))

(define a 23)
(define b 42)
(swap! a b)
```

Macro in Racket

```
(define-syntax-rule (min x y)
  (let ((x* x)
        (y* y))
    (if (< x* y*)
        x*
        y*)))
```

(LG 5-3) Hygiene

```
(define-syntax-rule (swap! x y)
  (let ((z x))
    (set! x y)
    (set! y z)))

(define a 15)
(define z 22)

(swap! a z)
```

Module System

```
#lang racket
(provide swap!)
(define-syntax-rule (swap! x y)
  (let ((z x))
    (set! x y)
    (set! y z)))
```

Import Macros

```
#lang racket
(require "swap.rkt")

(define a 15)
(define z 22)

(swap! a z)
```

Varargs

```
(define-syntax swap!
  (syntax-rules ()
    ((swap! x y)
     (let ((z x))
       (set! x y)
       (set! y z)))
    ((swap! x y z)
     (begin
       (set! x y)
       (set! y z)))))
```

Varargs

```
(define-syntax destructure
  (syntax-rules ()
    ((destructure exp (v1 ...) body)
     (apply (lambda (v1 ...)
              body)
            exp))))

(destructure (list 1 2 3) (a b c) (+ a b c))
```

Varargs

```
(define-syntax-rule (destructure exp (v1 ...) body)
  (let ((v exp))
    (destructure* v (v1 ...) body)))

(define-syntax destructure*
  (syntax-rules ()
    ((destructure* v () body) body)
    ((destructure* v (v1 v2 ...) body)
     (let ((v1 (car v))
           (rest (cdr v)))
       (destructure* rest (v2 ...) body)))))
```

Keywords in Patterns

```
(define-syntax if*
  (syntax-rules ()
    ((if* test then consequent else alternative)
     (if test consequent alternative))))

(if* (> a b) then 1 else 2)
(if* (> a b) else 1 then 2)
```

Literals in Macros

```
(define-syntax if*
  (syntax-rules (then else)
    ((if* test then consequent else alternative)
     (if test consequent alternative))))

(if* (> a b) then 1 else 2)
(if* (> a b) else 1 then 2)
; if*: bad syntax in: (if* (> a b) else 1 then 2)
```

(LG 2-1) Haskell List Comprehensions

```
let triangles =
      [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]

let rightTriangles =
      [ (a,b,c) |
        c <- [1..10],
        b <- [1..c],
        a <- [1..b],
        a^2 + b^2 == c^2 ]
```

Racket List Comprehensions

```
(define triangles
  (|| (list a b c)
      (<- c (from-to 1 10))
      (<- b (from-to 1 10))
      (<- a (from-to 1 10))))

(define right-triangles
  (|| (list a b c)
      (<- c (from-to 1 10))
      (<- b (from-to 1 c))
      (<- a (from-to 1 b))
      (= (+ (sqr a) (sqr b)) (sqr c))))
```

List Comprehensions im Haskell-Standard

Translation: List comprehensions satisfy these identities, which may be used as a translation into the kernel:

```
[ e | True ]              =  [e]
[ e | q ]                 =  [ e | q, True ]
[ e | b, Q ]              =  if b then [ e | Q ] else []
[ e | p <- l, Q ]         =  let ok p = [ e | Q ]
                                 ok _ = []
                             in concatMap ok l
[ e | let decls, Q ]      =  let decls in [ e | Q ]
```

where e ranges over expressions, p over patterns, l over list-valued expressions, b over boolean expressions, *decls* over declaration lists, q over qualifiers, and Q over sequences of qualifiers. `ok` is a fresh variable. The function `concatMap`, and boolean value `True`, are defined in the Prelude.

List Comprehensions in Racket

```
(define-syntax ||
  (syntax-rules (<- let)
    ((|| e #t) (list e))
    ((|| e q) (|| e q #t))
    ((|| e (<- p l) Q ...)
     (let ((ok
            (lambda (p)
              (|| e Q ...))))
       (concat-map ok l)))
((|| e (let decls) Q ...)
 (let decls
   (|| e Q ...)))
((|| e b Q ...)
 (if b
     (|| e Q ...)
     '())))
```

Syntax Objects as Values

```
#lang racket
(define-syntax my-case
  (lambda (x)
    (syntax-case x ()
      ((_ e c1 c2 ...)
       #`(let ((t e))
           #,(let f ((c1 #'c1) (cmore (syntax->list #'(c2 ...))))
               (if (null? cmore)
                   (syntax-case c1 (else)
                     ((else e1 e2 ...) #'(begin e1 e2 ...))
                     (((k ...) e1 e2 ...)
                      #'(when (memv t '(k ...)) (begin e1 e2 ...))))
                   (syntax-case c1 ()
                     (((k ...) e1 e2 ...)
                      #`(if (memv t '(k ...))
                            (begin e1 e2 ...)
                            #,(f (car cmore) (cdr cmore))))))))))))

(my-case 5
  ((1 2) 'one-two)
  ((3 4) 'three-four)
  ((5 6) 'five-six))
```

Exercise: Better Syntax

Imagine a more pleasant notation than Racket + Combinators from the Case Study chapter, provided it uses parentheses/brackets/curly braces.

Implement it!

DSL Evolution Strategy

1. combinator library
2. better syntax via macros
3. (optional) stand-alone syntax within Racket
4. stand-alone DSL within target infrastructure