CS4613 Lecture 5: Syntactic Sugar

David Bremner

January 25, 2024

Desugaring

p. 72

```
print([ x * x for x in range(0,10) ])
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
out = []
for x in range(0,10):
    out.append(x*x)
print(out)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

define-syntax-rule

We previously used define-syntax-rule to extend plait syntax.

```
(define-syntax-rule (let1 (var expr) body)
  (let ([var expr]) body))
```

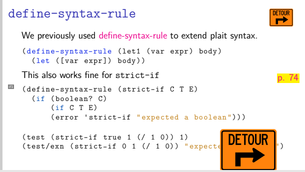
This also works fine for `strict-if`

p. 74

if1

```
(define-syntax-rule (strict-if C T E)
  (if (boolean? C)
      (if C T E)
      (error 'strict-if "expected a boolean")))

(test (strict-if true 1 (/ 1 0)) 1)
(test/exn (strict-if 0 1 (/ 1 0)) "expected a boolean")
```



define-syntax-rule

DETOUR

We previously used define-syntax-rule to extend plait syntax.

```
(define-syntax-rule (let1 (var expr) body)
  (let ([var expr]) body))
```

This also works fine for strict-if

p. 74

```
(define-syntax-rule (strict-if C T E)
  (if (boolean? C)
      (if C T E)
      (error 'strict-if "expected a boolean")))

(test (strict-if true 1 (/ 1 0)) 1)
(test/exn (strict-if 0 1 (/ 1 0)) "expecte          ")
```

DETOUR

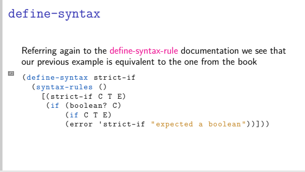
1. We switch to #lang racket here to follow the book, and because we need boolean?

define-syntax

Referring again to the define-syntax-rule documentation we see that our previous example is equivalent to the one from the book

if2

```
(define-syntax strict-if
  (syntax-rules ()
    [(strict-if C T E)
     (if (boolean? C)
         (if C T E)
         (error 'strict-if "expected a boolean"))]))
```



define-syntax

Referring again to the define-syntax-rule documentation we see that our previous example is equivalent to the one from the book

```
(define-syntax strict-if
  (syntax-rules ()
    [(strict-if C T E)
     (if (boolean? C)
         (if C T E)
         (error 'strict-if "expected a boolean"))]))
```

1. So far it isn't clear what the more general syntax buys us, but it will be when we look at recursive macros.
2. Try this in the macro stepper, but make sure you enable standard macro hiding

Let1 left-left-lambda

We just saw `let1` expanded to `let`. But if we have anonymous functions, we don't need `let` p. 76

let1

```
(define-syntax my-let1
  (syntax-rules ()
    [(my-let1 (var val) body)
     ((lambda (var) body) val)]))

(test (my-let1 (x 3) (+ x x)) 6)
```

The power of ...

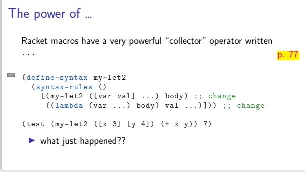
Racket macros have a very powerful “collector” operator written `...` p. 77

let2

```
(define-syntax my-let2
  (syntax-rules ()
    [(my-let2 ([var val] ...) body) ;; change
     ((lambda (var ...) body) val ...)])) ;; change

(test (my-let2 ([x 3] [y 4]) (+ x y)) 7)
```

- what just happened??



The power of ...

Racket macros have a very powerful "collector" operator written `...` p. 77

```
(define-syntax my-let2
  (syntax-rules ()
    [(my-let2 ([var val] ...) body) ;; change
     ((lambda (var ...) body) val ...)])) ;; change

(test (my-let2 ([x 3] [y 4]) (+ x y)) 7)
```

- what just happened??

1. The ... syntax is quite powerful (arguably a bit too much magic), and avoids most common needs for recursive macros

Recursive macros

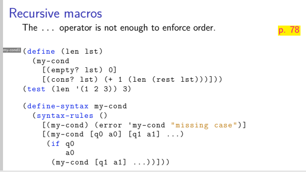
The ... operator is not enough to enforce order.

p. 78

my-cond1

```
(define (len lst)
  (my-cond
    [(empty? lst) 0]
    [(cons? lst) (+ 1 (len (rest lst)))]))
(test (len '(1 2 3)) 3)

(define-syntax my-cond
  (syntax-rules ()
    [(my-cond) (error 'my-cond "missing case")]
    [(my-cond [q0 a0] [q1 a1] ...)
     (if q0
         a0
      (my-cond [q1 a1] ...))]))
```



Recursive macros

The . . . operator is not enough to enforce order. p. 78

my-cond1
```
(define (len lst)
  (my-cond
    [(empty? lst) 0]
    [(cons? lst) (+ 1 (len (rest lst)))]))
(test (len '(1 2 3)) 3)

(define-syntax my-cond
  (syntax-rules ()
    [(my-cond) (error 'my-cond "missing case")]
    [(my-cond [q0 a0] [q1 a1] ...)
     (if q0
         a0
      (my-cond [q1 a1] ...))]))
```

1. For those familiar with functional programming, . . . behaves like `map`, expanding all of the clauses in parallel. What we need is more like `fold`
2. The error message is changed relative to the book. Why?

Missing case

my-cond2
```
(define (sign n)
  (my-cond
   [(< n 0) "negative"]
   [(> n 0) "positive"]))
(test/exn (sign 0) "missing")
```

- When is the error detected, at expansion time or at evaluation time?

Recursive binding

In let* bindings can refer to or shadow previous ones

```
(let* ([x 1]
       [y (+ x 2)]
       [x (+ y 3)])
  x)
```

This can be desugared to nested `let`

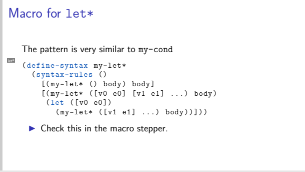
```
(let ([x 1])
  (let ([y (+ x 2)])
    (let ([x (+ y 3)])
      x)))
```

Macro for let*

The pattern is very similar to my-cond

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body) body]
    [(my-let* ([v0 e0] [v1 e1] ...) body)
     (let ([v0 e0])
       (my-let* ([v1 e1] ...) body))]))
```

- Check this in the macro stepper.



Macro for let*

The pattern is very similar to my-cond

let*

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body) body]
    [(my-let* ([v0 e0] [v1 e1] ...) body)
     (let ([v0 e0])
       (my-let* ([v1 e1] ...) body))]))
```

- Check this in the macro stepper.

1. Why is the base case not an error here?

Reducing repetition

We can replace the name of the macro with a wildcard _ , so we p. 79
can replace

```
(define-syntax unless
  (syntax-rules ()
    [(unless tst body ...)
     (if (not tst) (begin body ...) (void))]))

(define-syntax unless
  (syntax-rules ()
    [(_ tst body ...)
     (if (not tst) (begin body ...) (void))]))
```

Motivation

Consider using the `unless` macro where `not` is rebound.

rebnot

```
(let ([not (λ (v) v)])
  (unless false (println 1) (println 2)))
```

Naively this seems to expand to the following

```
(let ([not (λ (v) v)])
  (if (not false)
      (begin (println 1) (println 2))
      (void)))
```

This would be bad, but luckily something else happens.

p. 80

Coloured Code (hygiene)

If we check our example in the macro stepper

p. 80

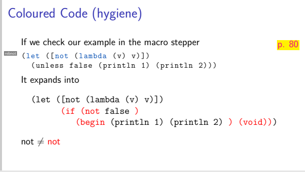
rebnot

```
(let ([not (lambda (v) v)])
  (unless false (println 1) (println 2)))
```

It expands into

```
  (let ([not (lambda (v) v)])
        (if (not false )
           (begin (println 1) (println 2) ) (void)))
```

not $\neq$ not



Coloured Code (hygiene)

If we check our example in the macro stepper

p. 80

rebnot

```
(let ([not (lambda (v) v)])
  (unless false (println 1) (println 2)))
```

It expands into

```
  (let ([not (lambda (v) v)])
        (if (not false )
           (begin (println 1) (println 2) ) (void)))
```

not $\neq$ not

1. Note our colors (and the macro steppers) differ from the book

Double evaluation

Consider the following use of `if` to make a short circuiting `or`. p. 82

```
(define-syntax or-2
  (syntax-rules ()
    [(_ e1 e2)
     (if e1 e1 e2)]))
```

This evaluates e1 twice, which is potentially expensive, or worse

or1

```
(test (or-2 (member 'y '(x y z)) "not found") '(y z))
(define v #f)
(define (toggle) (set! v (not v)) v) ;; side-effect
(test (or-2 (toggle) 'else) #t)
```

local binding in macros

We can avoid double evaluation in general using local binding.
Thanks to hygiene we don't have to worry about the variable name

or2

```
(define-syntax or-2
  (syntax-rules ()
    [(_ e1 e2)
     (let ([v e1])
       (if v v e2))]))
```

- How do we know this avoids double evaluation?

Another hygiene example

or3

```
(let ([v 1])
  (or-2 false v))
```

If we rename the v's by defining context, the expansion looks like

```
(let ([v0 1]) (let ([v1 false]) (if v1 v1 v0)))
```

In the macroexpander, we get

```
(let ([v 1]) (let ([v false]) (if v v v)))
```

v0 =v, v1 = v