Racket: Macros

Advanced Functional Programming

Kostis Sagonas

November 2019

(original set of slides by Jean-Noël Monette)

Content

- Macros
  - Pattern-based macros
  - Hygiene
  - Syntax objects and general macros
  - Examples

Macros

(According to the Racket Guide...) A macro is a new syntactic form that can be expanded into existing forms through a transformer.

In general, it is a way to extend the language with new constructs.

Let's start with a simple example. Say we do not have the `or` construct and we want to implement it in terms of `if`.

Implementing or: First try

```
(define (or x y)
  (if x x y))
```

Implementing or: First try

```
(define (or x y)
  (if x x y))
```

What are the problems?

Implementing or: First try

```
(define (or x y)
  (if x x y))
```

What are the problems?

```
(or 3 (/ 2 0))
```

Implementing or: First try

```
(define (or x y)
  (if x x y))
```

What are the problems?

```
(or 3 (/ 2 0))
```

- `y` is always evaluated.

Implementing or: First try

```
(define (or x y)
  (if x x y))
```

What are the problems?

```
(or 3 (/ 2 0))
```

- `y` is always evaluated.

A function does not work. We need something that would transform the `or` into an `if` before run time.

Using define-syntax-rule

define-syntax-rule defines a macro.

```
(define-syntax-rule (or x y)
    (if x x y))
```

(or 3 (/ 2 0)) is effectively transformed into **(if 3 3 (/ 2 0))**.

Using define-syntax-rule

`define-syntax-rule` defines a macro.

```
(define-syntax-rule (or x y)
    (if x x y))
```

`(or 3 (/ 2 0))` is effectively transformed into `(if 3 3 (/ 2 0))`.

This problem is solved. But we introduced another one:

```
(or (begin (display 3) 3) (/ 2 0))
```

`x` is evaluated twice.

Introducing let

```
(define-syntax-rule (or x y)
  (let ([z x])
    (if z z y)))

(or 3 (/ 2 0))
(or (begin (display 3) 3) (/ 2 0))
```

Introducing let

```
(define-syntax-rule (or x y)
  (let ([z x])
    (if z z y)))

(or 3 (/ 2 0))
(or (begin (display 3) 3) (/ 2 0))
```

Seems like we are good.

Hygiene

Consider another example

```
(define-syntax-rule (swap! x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

Hygiene

Consider another example

```
(define-syntax-rule (swap! x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

What if we pass a variable named `tmp` (or even `set!` or `let`) to `swap!`?

Hygiene

Consider another example

```
(define-syntax-rule (swap! x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

What if we pass a variable named `tmp` (or even `set!` or `let`) to `swap!`?

The naive rewriting would produce an erroneous code.

Hopefully, the Racket macro system is "hygienic"; that is, it makes sure to avoid clashes between identifiers and it uses lexical scoping.

More complicated macros

```
(define-syntax or
  (syntax-rules ()
    [(or) #f]
    [(or x) x]
    [(or x y) (let ([z x])
                (if z z y))]
    [(or x y ...) (or x (or y ...))]))
```

- **`syntax-rules`** defines several rules by pattern matching.
- **`...`** in the pattern matches several (zero, one or more) occurences of the last pattern (**`y`** in this case).
- **`...`** in the template provides all the bound occurences.

Using literal keywords

```
(define-syntax while
  (syntax-rules (do)
    [(while cond do body ...)
      (let loop ()
        (when cond
          body ...
          (loop)))]))

(while (> x 0) do
       (displayln x)
       (set! x (- x 1)))
```

do is part of the macro definition.

A wrong version of while

```
(define-syntax while
  (syntax-rules (do)
    [(while cond do body ...)
     (when cond
       body ...
       (while cond do body ...))]))
```

Why?

A wrong version of while

```
(define-syntax while
  (syntax-rules (do)
    [(while cond do body ...)
     (when cond
       body ...
       (while cond do body ...))]))
```

Why?

The macro are expanded before runtime.

In this case, the expansion never completes.

Examples

- Memoization
- Rewriting

Example: Implementing Memoization

Memoization is to record the result of previous calls to a procedure to avoid recomputing them again.

For instance, a (naive) fibonacci procedure takes exponential time to complete.

```
(define (fib x)
  (case x
    [(0 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

Memoization would allow us to reuse already computed results.

We will define a macro **`define-memo`** to define a procedure with memoization.

Note: memoization only makes sense for procedures that are really functional, i.e., without side-effects.

define-memo

```
(define-syntax-rule (define-memo (f args ...) body ...)
  (define f
    (let ([memo (make-hash)])
      (lambda (args ...)
        (cond [(hash-has-key? memo (list args ...))
               (hash-ref memo (list args ...))]
              [else
               (let ([res (begin body ...)])
                 (hash-set! memo (list args ...) res)
                 res)])))))
```

Previous results are stored in a (mutable) hash-table.

The keys are the arguments of the function.

The actual body of the function is executed only if the arguments are not already in the hash-table.

define-memo, application

```
(define-syntax-rule (define-memo (f args ...) body ...)
  (define f
    (let ([memo (make-hash)])
      (lambda (args ...)
        (cond [(hash-has-key? memo (list args ...))
               (hash-ref memo (list args ...))]
              [else
               (let ([res (begin body ...)])
                 (hash-set! memo (list args ...) res)
                 res)])))))

(define-memo (fib/m x)
  (display x) ; side effect to see when we actually execute the body
  (case x
    [(0 1) 1]
    [else (+ (fib/m (- x 1)) (fib/m (- x 2)))]))

(fib/m 10)
(fib/m 10)
```

Example: Simplifying expressions

It is often useful to simplify expressions.

We will do it here by rewriting rules.

Consider this target piece of code

```
(define s (simplifier
            [(+ ,x 0) -> ,x]
            [(* ,x 1) -> ,x]
            [(+ ,x ,x) -> (* ,x 2)]))

(s '(+ (+ y 0) (* y 1)))
```

`simplifier`, which is defined on the next two slides, returns a function of one argument: the expression to simplify.

Simplifying one expression

```
(define-syntax simplifier
  (syntax-rules (->)
    [(simplifier (x -> y) ...)
     (lambda (form)
       (match form
         [`x `y]
         ...
         [else form]))]))
```

Each rule is rewritten as a clause in a `match` expression.

The whole code is enclosed in a procedure.

`->` is part of the macro definition, not a variable.

Making it recursive

The previous code only simplifies the root of the expression.

We need to call the simplification on each sub-expression.

We need to repeat the simplification until fix-point.

Making it recursive

The previous code only simplifies the root of the expression.

We need to call the simplification on each sub-expression.

We need to repeat the simplification until fix-point.

```
(define ((recur simplify) form)
  (let loop ([reduced-form
              (cond [(list? form)
                     (map (recur simplify) form)]
                    [else form])])
    (let ([final-form (simplify reduced-form)])
      (if (equal? final-form reduced-form)
          final-form
          (loop final-form)))))
```

`simplify` is the function produced by `simplifier`.

Putting it together

```
(define-syntax simplifier
  (syntax-rules (->)
    [(simplifier (x -> y) ...)
     (recur (lambda (form) ; Added recur here
              .....))])) ; .....: the same as before

(define s (simplifier
           [(+ ,x 0) -> ,x]
           [(* ,x 1) -> ,x]
           [(+ ,x ,x) -> (* ,x 2)]))

(s '(+ (+ y 0) (* y 1)))
```

Going further

Up to now, we have seen macros based on patterns, rewritten to a template.

Macros can do more than that, by manipulating the syntax (almost) however you want.

For this, we need to introduce syntax objects.

Syntax objects

`(syntax (+ 1 2))` returns a syntax object representing the data `(+ 1 2)` with lexical information and source code location.

Syntax objects can be manipulated by the macro system.

Syntax objects

`(syntax (+ 1 2))` returns a syntax object representing the data `(+ 1 2)` with lexical information and source code location.

Syntax objects can be manipulated by the macro system.

```
(define-syntax Hello
  (lambda (stx)
    (display stx)
    (datum->syntax stx (cdr (syntax->datum stx)))))
(Hello + 1 3)
```

The body of `define-syntax` must be a procedure taking a syntax object and returning a syntax object.

syntax-case

`syntax-rules` is a shortcut for "simple" macros: one writes directly the resulting code.

In the general form, one can use the whole power of Racket to modify the syntax.

`syntax-case` allows one to match the syntax against patterns.

`#'` is a shortcut for **`syntax`**

There is also **`#``** and **`#,`** for quasisyntax and unsyntax.

syntax-case: example

The swap example with syntax.

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap a b)
       #'(let ([tmp a])
           (set! a b)
           (set! b tmp))])))
```

syntax-case: example

The shortcut version to avoid the lambda.

```
(define-syntax (swap stx)
  (syntax-case stx ()
    [(swap a b)
     #'(let ([tmp a])
         (set! a b)
         (set! b tmp))]))
```

syntax-case: example

The shortcut version to avoid the lambda.

```
(define-syntax (swap stx)
  (syntax-case stx ()
    [(swap a b)
     #'(let ([tmp a])
         (set! a b)
         (set! b tmp))]))
```

This is not better than **`syntax-rules`** but wait...

syntax-case: example (2)

Checking (before runtime) that the arguments make sense

```
(define-syntax (swap stx)
  (syntax-case stx ()
    [(swap a b)
     (cond [(and (identifier? #'a) (identifier? #'b))
            #'(let ([tmp a])
                (set! a b)
                (set! b tmp))]
           [else (raise-syntax-error
                  #f
                  "not an identifier"
                  stx
                  (if (not (identifier? #'a))
                      #'a
                      #'b))])]))
```

More syntax manipulation

`syntax-e` returns the data in the syntax object.

`syntax->datum` recursively calls **`syntax-e`**.

`datum->syntax` creates a syntax object from some data and the lexical information of another syntax object.

`syntax-list` is like **`syntax-e`** but it unrolls a whole list.

`with-syntax` is similar to a **`let`** statement but for syntax objects.

`syntax?` tells whether the given value is a syntax object.

`identifier?` tells if the syntax object represents an identifier (**`syntax-e`** returns a symbol).

Example: extended rewriting

Now we want to be able to put guards on our rewriting rules.

```
(define s (simplifier
           [(+ ,x 0) -> ,x]
           [(* ,x 1) -> ,x]
           [(+ ,x ,x) -> (* ,x 2)]
           [(+ ,k1 ,k2)
            ? (and (number? k1) (number? k2))
            -> ,(+ k1 k2)]))

(s '(+ (+ y (+ 2 -2)) (* y 1)))
```

Extended rewriting with syntax

```
(define-syntax (simplifier stx)
  (define (clause expr)
    (syntax-case expr (? ->)
      [(x -> y) #'[`x `y]]
      [(x ? z -> y) #'[`x (=> fail)
                          (when (not z) (fail)) `y]]))
  (syntax-case stx ()
    [(_  expr ...)
     (with-syntax ([(res ...)
                    (map clause (syntax->list
                                 #'(expr ...)))])
       #'(simplify (lambda (form)
                     (match form
                       res
                       ...
                       [else form]))))]))
```

Breaking the hygiene

Sometimes we want to create identifiers that are available outside of the macro.

This happens when creating a struct, for instance.

`(struct box (val))` creates e.g. the procedures `box-val` and `box?`.

Breaking the hygiene

Sometimes we want to create identifiers that are available outside of the macro.

This happens when creating a struct, for instance.

`(struct box (val))` creates e.g. the procedures `box-val` and `box?`.

There is a simple way to break hygiene:

Use `(datum->syntax lex-cont name)` where `lex-cont` is an identifier that is given to the macro.

In that case, the new name inherits the same lexical scope.

Breaking the hygiene: example

Define a for loop with `it` bound to the current value.

```
(define-syntax (for stx)
  (syntax-case stx (do)
    [(for (obj1 objs ...) do body ...)
     (with-syntax ([it (datum->syntax #'obj1 'it)])
       #'(let loop ([its (list objs ...)]
                    [it obj1])
           body ...
           (when (not (empty? its))
             (loop (cdr its) (car its)))))]))

(for (1 2 3 4) do (display it)) ; OK
(for (1 2 3 4) do (display its)) ; Does not work
```

Breaking the hygiene: building names

This piece of code shows how to build a name from another one:

```
(datum->syntax
 #'orig-name
 (string->symbol
  (string-append "prefix-"
                 (symbol->string
                  (syntax->datum #'orig-name)))))
```

One need to go from a syntax object to a symbol to a string, manipulate the string, and then all the other way around.

Phases

We can use normal procedures when defining macros.

But macros are expanded before runtime.

How is that possible?

Phases

We can use normal procedures when defining macros.

But macros are expanded before runtime.

How is that possible?

There are several phases.

The identifiers in each phase are only available in that phase, unless imported.

One can have even more than two phases.

Conclusion: word of caution

Macros can be very powerful.

As any powertool, they must be used with care.

In particular, do not use a macro if a procedure can do the work.