CSE341: Programming Languages

Lecture 16
Macros

Dan Grossman
Fall 2011

This lecture

- What are macros
- Why are macros difficult to use sensibly
- Using Racket's macro system
  - Defining macros
  - Watching out for evaluation order and (re)-evaluation
  - Why *hygiene* makes Racket's macros much easier to use sensibly
- When (not) to use macros

What is a macro

- A *macro* describes how to transform some new syntax into different syntax in the source language
- A macro is one way to implement syntactic sugar
  - "Replace any syntax of the form `e1 andalso e2` with `if e1 then e2 else false`"
- A *macro system* is a language (or part of a larger language) for defining macros
- *Macro expansion* is the process of rewriting the syntax to eliminate macro uses
  - Before a program is run (or even compiled)

Tokenization

First question for a macro system: How does it tokenize?

- Macro systems generally work at the level of *tokens* not sequences of characters
  - So must know how programming language tokenizes text
- Example: "replace all occurrences of `car` with `hd`"
  - Would not rewrite `(+ cart foo)` to `(+ hdt foo)`
  - Would not rewrite `car-door` to `hd-door`
    - But would in C where `car-door` is subtraction

Parenthesization

Second question for a macro system: How does associativity work?

C/C++ basic example:

```
#define ADD(x,y) x+y
```

Probably *not* what you wanted:

`ADD(1,2/3)*4` means `1+2/3*4` not `(1+2/3)*4`

So C macro writers use lots of parentheses, which is fine:

```
#define ADD(x,y) ((x)+(y))
```

Racket won't have this problem:

- Macro use: `(macro-name …)`
- After expansion: `( something else in same parens )`

Local bindings

Third question for a macro system: Can variables shadow macros?

Suppose macros also apply to variable bindings. Then:

```
(let  ([hd 0][car 1]) hd) ; 0
(let* ([hd 0][car 1]) hd) ; 0
```

Would become:

```
(let  ([car 0][car 1]) car) ; error
(let* ([car 0][car 1]) car) ; 1
```

This is why C/C++ convention is all-caps macros and non-all-caps for everything else

Racket gets this and other scope gotchas "right"

Example Racket macro definitions

Two simple macros

```
(define-syntax my-if              ; macro name
  (syntax-rules (then else)       ; other keywords
    [(my-if e1 then e2 else e3)   ; macro use
     (if e1 e2 e3)]))             ; form of expansion
```

```
(define-syntax comment-out        ; macro name
  (syntax-rules ()                ; other keywords
    [(comment-out ignore instead) ; macro use
     instead]))                   ; form of expansion
```

If the form of the use matches, do the corresponding expansion

- In these examples, list of possible use forms has length 1
- Else syntax error

Example uses

It's like we added keywords to our language

- Other keywords only keywords in uses of that macro
- Syntax error if keywords misused
- Rewriting ("expansion") happens before execution

```
(my-if x then y else z)   ; (if x y z)
(my-if x then y then z)   ; syntax error
(my-if x then (begin (print "hi") 34) then 15)

(comment-out (begin (print "hi") 34) 15)
(comment-out (car null) #f)
```


Revisiting delay and force

Recall our definition of promises from last lecture

- Should we use a macro instead to avoid clients' explicit thunk?

```
(define (my-delay th)
  (mcons #f th))

(define (my-force p)
  (if (mcar p)
      (mcdr p)
      (begin (set-mcar! p #t)
             (set-mcdr! p ((mcdr p)))
             (mcdr p))))
```

```
(f (my-delay (lambda () e)))
```

```
(define (f p)
  (… (my-force p) …))
```


A delay macro

- A macro can put an expression under a thunk
  - Delays evaluation without explicit thunk
  - Cannot implement this with a function
- Now client then should *not* use a thunk (that would double-thunk)
  - Racket's pre-defined **`delay`** is a similar macro

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda() e))]))
```

```
(f (my-delay e))
```


What about a force macro?

We could define **`my-force`** with a macro too

- Good macro style would be to evaluate the argument exactly once (use **`x`** below, not multiple evaluations of **`e`**)
- Which shows it is bad style to use a macro at all here!
- Don't use macros when functions do what you want

```
(define-syntax my-force
  (syntax-rules ()
    [(my-force e)
     (let([x e])
       (if (mcar x)
           (mcdr x)
           (begin (set-mcar! x #t)
                  (set-mcdr! p ((mcdr p)))
                  (mcdr p))))]))
```


Another bad macro

Any *function* that doubles its argument is fine for clients

```
(define (dbl x) (+ x x))
(define (dbl x) (* 2 x))
```

- These are equivalent to each other

So macros for doubling are bad style but instructive examples:

```
(define-syntax dbl (syntax-rules()[(dbl x)(+ x x)]))
(define-syntax dbl (syntax-rules()[(dbl x)(* 2 x)]))
```

- These are not equivalent to each other. Consider:

```
(dbl (begin (print "hi") 42))
```

More examples

Sometimes a macro *should* re-evaluate an argument it is passed
- If not, as in **dbl**, then use a local binding as needed:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x)
      (let ([y x]) (+ y y))]))
```

Also good style for macros not to have surprising evaluation order
- Good rule of thumb to preserve left-to-right
- Bad example (fix with a local binding):

```
(define-syntax take
  (syntax-rules (from)
     [(take e1 from e2)
      (- e2 e1)]))
```


Local variables in macros

In C/C++, defining local variables inside macros is unwise
- When needed done with hacks like **__strange_name34**

Here's why with a silly example:
- Macro:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x) (let ([one 1])
                (* 2 x one))]))
```

- Use:

```
(let ([one 7]) (dbl one))
```

- Naïve expansion:

```
(let ([one 7]) (let* ([one 1])
                 (* 2 one one)))
```

- But instead Racket "gets it right," which is part of *hygiene*

The other side of hygiene

This also looks like it would do the "wrong" thing
- But Racket's hygienic macros do the "right thing"

- Macro:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x) (* 2 x)]))
```

- Use:

```
(let ([* +]) (dbl 42))
```

- Naïve expansion:

```
(let ([* +]) (* 2 42))
```


How hygienic macros work

A hygienic macro system:
- Secretly renames local variables in macros with fresh names
- Looks up variables used in macros where the macro is defined

Neither of these rules are followed by the "naïve expansion" most macro systems use
- Without hygiene, macros are much more brittle (non-modular)

Rarely hygiene is not what you want
- Racket has somewhat complicated support for that

More examples

See **lec16.rkt** for macros that:

- Allow 0, 1, or 2 local bindings with fewer parens than **let***
- A for loop for executing a body a fixed number of times
- A re-implementation of **let*** in terms of **let**
  - Requires macros that take any number of arguments
  - Requires recursive macros