

CS450

Structure of Higher Level Languages

Lecture 23: Macros

Tiago Cogumbreiro

Today we will...

- Why macros are needed?
- Where are macros used?
- Safe versus unsafe macros
- The problems of using macros
- Macros in Racket
- Macros: side-effects
- Macros: controlling evaluating
- Macros: types in macros
- Macros: pattern matching

Acknowledgment: Today's lecture is inspired by Professor Dan Grossman's wonderful lecture [in CSE341](#) from the University of Washington.



Macro systems

What is a macro

A macro is a technique to perform reusable source-code transformations with the objective to extend the language semantics.

- **Macro definition:** describes how the transformation occurs
- **Macro system:** the language used to describe transformations
- **Macro expansion:** the process of transforming the syntax according to some macro

Macro expansion occurs before the program is run (and compiled).

Macros in Racket

Macros in Racket are used as function calls, however evaluation does **not** proceed as it does with a function application.

Example 1

Expands a do-macro that accepts special keywords/symbols

```
(do x ← (push 10) (pop))
```

into

```
(bind (push 10) (lambda (x) (pop)))
```

Example 2

Omit some expressions of the macro

```
(comment-out (/ x 0) 10)
```

expands into

```
10
```

Example uses

Macros can vastly transform the Racket language

Macros can:

- encode infix notation
- encode alternate evaluation methods (such as lazy evaluation)
- generate boilerplate code (repetitive code)
- encode different programming models (succinct syntax for monads, OOP, etc)

Macros uses in practice

- Most Racket's language features are built with macros!
Examples: `cond`, promises, OOP system, etc
- Automatic JSON/XML serialization in OCaml
- Boilerplate generation (bridges) from OCaml to JavaScript, and from Rust to GLib (C-based OOP runtime)

The perils of macros

The perils of macros

- **Unclear computational model:**
How are the parameters evaluated? Does the macro produce side effects?
- **Limited composability:**
Is the result of a macro a value? can it be passed around?
- **Stack-trace obfuscation:**
The emitted code may generate a non-obvious stack trace, which hinders debugging.
- **Non-terminating compilation:**
Most macros-systems are Turing complete, which means they may not terminate. They may slow down compilation times, a problem at scale.

Declare macros sparingly and with caution

Following we will learn...

- Manipulating syntactic elements (tokens, parentheses, scope)
- Defining macros
- Controlling expression evaluation
- Introduce macro *hygiene*

Macros manipulate syntactic terms

- A macro system usually operates on the **concrete** syntax
- Recall our exercises on datums, a macro system operates at the datums level.
- In the concrete syntax, there will be some notion of a literal, an identifier, a sequence, a datum, maybe control-flow data structures
- Generally, a macro system does **not** operate at the lexical level
For example, a macro system cannot declare a new parsing rule to recognize, say, binary number literals.

Macro expansion

How macro systems generate code?

Does the macro system support structured data?

Unstructured expansion

The C macro system operates at the textual level, there is no notion of structure, and simply allows for free-text transformation.

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

Expression `ADD(1, 2) * 3` expands to `1 + 2 * 3` and not to `(1 + 2) * 3`.

Structured expansion

The Racket macro system operates at the concrete syntax level, so code transformations retain their structure.

```
(define-syntax-rule (ADD x y) (+ x y))  
(check-equal? (* (ADD 1 2) 3) 9)
```

C: The perils of unstructured macros

"What is the worst real-world macros/pre-processor abuse you've ever come across?"
Stack Overflow.

```
int foo(state_t *state) {
    int a, b, rval;

    $
    if (state->thing == whatever) {
        $
        do_whatever(state);
    }
    // more code

    $
    return rval;
}
```

```
#if DEBUG
#define $ log("%s %d", __FILE__, __LINE__);
#else
#define $
#endif
```

Source: Frank Szczerba

The infamous UNIX Bourne Shell

```
#define IF    if (
#define THEN ) {
#define ELSE } else {
#define ELIF } else if (
#define FI   ; }

VOID    free(ap)
    BLKPTR    ap;
{
    REG BLKPTR    p;

    IF (p=ap) ANDF p<bloktop
    THEN    Lcheat((--p)→word) &= ~BUSY;
    FI
}
```

The source code of the UNIX Bourne shell (1970) used macros to make C code more similar to Algol 68. Source code available online: macros defined in [mac.h](#), example program [blok.c](#).

Source: [Jim Ferrans](#)

The Love/Hate Relationship with the C Preprocessor

The Love/Hate Relationship with the C Preprocessor: An Interview Study. Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. ECOOP, 2015.

Why use macros

- portability: support different operating systems with little change
- variability: removing parts of the library to reduce the binary code size

```
if (b_ffname ≠ NULL
#ifdef FEAT_NETBEANS
    && netbeansReadFile
#endif
) {
    // code
}
```

```
mfp = open(mf_fname
#ifdef UNIX
    , (mode_t)0600
#endif
#ifdef MSDOS
    , S_IREAD | S_IWRITE
#endif
);
```

```
#if defined(GUI_W32)
void msgNetbeansW32(
#else
void msgNetbeans(Xt client,
#endif
XtInputId *id) {
    // code
}
```

Code snippets from the Vim editor.



Macros in Racket

A macro example

Use `define-syntax-rule` as you would use a `define`.

```
(define-syntax-rule (ADD x y)
  (+ x y))
(check-equal? (* (ADD 1 2) 3) 9)
```

Side effects

keeping in mind that its contents are *not* evaluated. The contents of the macro are therefore **inlined**.

Example

```
(define-syntax-rule (SQR x)
  (* x x))
```

Beware of side-effects!

```
; Prints !!
(define (f) (display "!") 3)
(SQR (f))
```

Spec

```
(check-equal?
  (SQR (* 2 3))
  (* (* 2 3) (* 2 3))) ; expands x twice!
```

Solution

```
(define-syntax-rule (SQR x)
  ((lambda (new-x) (* new-x new-x))
   x))
; Or, use the let construct
(define-syntax-rule (SQR x)
  (let ([new-x x]) (* new-x new-x)))
```

Why would you
want to control evaluation?

Controlling evaluation: example 1

Macros allow us to control evaluation, which lets us delay evaluation. Here is an implementation of an `if` command.

```
(define-syntax-rule (IF cnd then-branch else-branch)
  (or (and cnd then-branch) else-branch))
; Sanity tests; in case of eager evaluation it should crash
(check-equal? (IF #t 1 (/ 1 0)) 1)
(check-equal? (IF #f (/ 1 0) 2) 2)
```

Controlling evaluation: example 2

When creating a testing library, we may need to show the user which code is failing. We can quote a macro variable and print the datum.

```
(define-syntax-rule (assert x)
  (IF x (void) (error "Condition failed: " (quote x))))

(assert (and #f 10))
; Condition failed: (and #f 10) [,bt for context]
```

Controlling evaluation: example 3

```
(define-syntax-rule (letin x v e)
  ((lambda (x) e) v))

(check-equal? (letin x (+ 10 50) x) 60)
```

Adding types to macros

Restricting what appears where

The macro construct `define-simple-macro` allows restricting what *kind* of parameter is expected, which improves the error messages.

Version 1

```
(require syntax/parse/define)
(define-simple-macro (fn x body)
  (lambda (x) body))

(check-equal? ((fn x x) 10) 10)
; (fn 11 10)
; lambda: not an identifier, identifier with
; default, or keyword
; at: 11
; in: (lambda (11) 10)
; [,bt for context]
```

Version 2

```
(require syntax/parse/define)
(define-simple-macro (fn x:id body:expr)
  (lambda (x) body))

(check-equal? ((fn x x) 10) 10)
; (fn 11 10)
; fn: expected identifier
; at: 11
; in: (fn 11 10)
; [,bt for context]
```


Introducing syntactic literals

```
(define-simple-macro (fn x (~literal →) expr)  
  (lambda (x) expr))
```

```
(check-equal? ((fn x → x) 10) 10)
```

Pattern matching in macros

Revisiting the `do` notation

```
(define-syntax do
  (syntax-rules (←) ; here we declare reserved syntactic tokens
    ; Only one monadic-op, return it
    [(_ mexp) mexp] ; alternatively, we could write (do mexp)
    ; A binding operation
    [(_ var ← mexp rest ...) (bind mexp (lambda (var) (do rest ...)))]
    ; No binding operator, just ignore the return value
    [(_ mexp rest ...) (bind mexp (lambda (-) (do rest ...)))]))
```

Homework Assignment 7

Homework Assignment 7

The interpreter

1. Use `do`, `eff-bind`, and `eff-pure`
2. Use `match` instead of `cond` and `lambda-args`

Handling multiple arguments

1. Function applications
2. Function declarations

Supporting primitives

1. `if`
2. `builtin`

Homework Assignment 7

The interpreter

1. The memory parameter and passing memory around must be abstract away via monads (`do`, `eff-bind`, `eff-pure`). **Start by this one!**
2. Use pattern matching instead of accessors `s:define-var`, `s:define-body`, `s:seq-fst`, `s:seq-snd`
3. Use `match` instead of `cond`
4. If you decide not to submit HW5 again, I can give you a solution of HW5

Handling multiple parameters

Function declaration

```
(lambda (x y z) z) → (lambda (x) (lambda (y) (lambda (z) z)))  
(lambda () 10)   → (lambda (-) 10)
```

Function application

```
(f 1 2 3) → (((f 1) 2) 3)  
(f)      → (f (void))
```

Supporting primitives

Branching support (if)

■ The `if` expects 3 parameters (curried); we follow Racket's rules to to

$$\frac{e_c \Downarrow_E \#f \quad \blacktriangleright \quad e_f \Downarrow v_f}{(((\text{if } e_c) e_t) e_f) \Downarrow_E v_t} \text{ (E-if-f)}$$

$$\frac{e_c \Downarrow_E v \quad v \neq \#f \quad \blacktriangleright \quad e_t \Downarrow v_t}{(((\text{if } e_c) e_t) e_f) \Downarrow_E v_f} \text{ (E-if-t)}$$

Example

```
(((if x) true-branch) else-branch)
```


Supporting primitives

Built-ins support

You will need to extend the function application rule and check if the result of evaluating e_f is either a `closure` or a `builtin`. If it is the former, then evaluate the function application as usual. If it is the latter, then evaluate the function application as described below.

$$\frac{e_f \Downarrow_E (\mathbf{builtin} f) \quad e_a \Downarrow_E v_a}{(e_f e_a) \Downarrow_E f(v_a)} \quad (\mathbf{E-app-b})$$