

# **Programming Abstractions**

## **Lecture 29: More macros**

**Stephen Checkoway**

# Announcements

Office hours Tuesday 13:30–14:30

Exam 2 graded

# Scheme macros

```
(define-syntax keyword
  (syntax-rules ()
    [pattern-1 transformation-1]
    [pattern-2 transformation-2]
    ...
    [pattern-n transformation-n]))
```

Patterns can specify variables that can be used in the corresponding transformation

# Potential confusion in these slides

**Please ask if it's not clear!**

Sometimes I use `...` to indicate that I've omitted something like

- `(foo arg-1 arg-2 ... arg-n)`
- `(cond [(lit-exp? exp) ...]  
          [(var-exp? exp) ...]  
          ...  
          [else ...])`

Inside macro patterns `...` means to match the preceding item 0 or more times

Inside macro transformation `...` means to perform the transformation to each matched item

- `(syntax-rules (  
    [(_ foo [bar oof] ...)  
      (foo (list bar ...) oof ...)])`

# Consider switch

```
(switch exp [case-1 exp-1] ... [case-n exp-n])
```

The behavior we want is

- exp is evaluated;
- the result is compared against each of case-1 through case-n in order;
- if the result is equal to case-i then the value of the expression is exp-i

It should behave the same as

```
(let ([result exp])  
  (cond [(equal? result case-1) exp-1]  
        ...  
        [(equal? result case-n) exp-n]))
```

# Let's define a switch syntax!

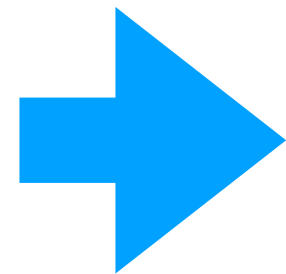
```
(define-syntax switch
  (syntax-rules ()
    [ ( _ exp [case case-exp] ...)
      (let ([result exp])
        (cond [(equal? result case) case-exp] ...))]))
```

```
(switch (- 2 1)
  [0 "zero"]
  [1 "one"]
  [2 "two"])
```

# Let's define a switch syntax!

```
(define-syntax switch
  (syntax-rules ()
    [ ( _ exp [case case-exp] ... )
      (let ([result exp])
        (cond [(equal? result case) case-exp] ...))]))
```

```
(switch (- 2 1)
  [0 "zero"]
  [1 "one"]
  [2 "two"])
```



```
(let ([result (- 2 1)])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"])))
```

What is the value of this?

```
(define-syntax switch
  (syntax-rules ()
    [(_ exp [case case-exp] ...)
     (let ([result exp])
       (cond [(equal? result case) case-exp] ...))]))
```

```
(switch 3
  [0 "zero"]
  [1 "one"]
  [2 "two"])
```

A. 3

B. "three"

C. void

D. It's an error



# Let's add an `[else exp]` to `switch`

We want to support an `else`

```
(switch 3
  [0 "zero"]
  [1 "one"]
  [2 "two"]
  [else "something else"])
```

As we've currently implemented `switch`, this won't work

- Why not?

# Let's add an `[else exp]` to `switch`

We want to support an `else`

```
(switch 3
  [0 "zero"]
  [1 "one"]
  [2 "two"]
  [else "something else"])
```

As we've currently implemented `switch`, this won't work

► Why not?

```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"]
        [(equal? result else) "something else"])))
```

# First attempt

```
(define-syntax switch
  (syntax-rules ()
    [ (_ exp [case case-exp] ... [else else-exp])
      (let ([result exp])
        (cond [(equal? result case) case-exp] ...
              [else else-exp]))]
    [ (_ exp [case case-exp] ...)
      (switch exp [case case-exp] ... [else (void)])]))
```

Recursive  
macros are  
fine!

Two rules, each with a **pattern** and a matching **transformation**

Idea: a (switch ...) without an [else ...] matches the second rule;  
a (switch ...) with an [else ...] matches the first rule

# Trying it out

```
(switch 3
  [0 "zero"]
  [1 "one"]
  [2 "two"]
  [else "something else"])
```

returns "something else"

Success?

# Not quite

```
(switch 3  
  [0 "zero"]  
  [1 "one"]  
  [2 "two"])
```

returns "two"!

The problem is this `switch` matches the first pattern

```
(_ exp [case case-exp] ... [else else-exp])
```

We need to inform Racket that `else` is not a pattern variable and is meant to be matched literally

# Not quite

```
(switch 3
  [0 "zero"]
  [1 "one"]
  [2 "two"])
```

```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [2 "two"])))
```

returns "two"!

The problem is this `switch` matches the first pattern

```
(_ exp [case case-exp] ... [else else-exp])
```

We need to inform Racket that `else` is not a pattern variable and is meant to be matched literally

# Literal matches

```
(syntax-rules (literal ...) [pattern transform] ...)
```

The first argument to `syntax-rules` is a list of words to match literally

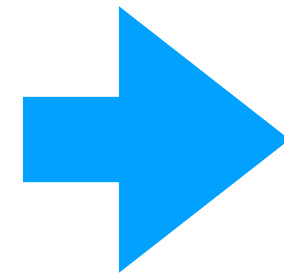
```
(define-syntax switch
  (syntax-rules (else)
    [(_ exp [case case-exp] ... [else else-exp])
     (let ([result exp])
       (cond [(equal? result case) case-exp] ...
             [else else-exp]))]
    [(_ exp [case case-exp] ...)
     (switch exp [case case-exp] ... [else (void)]))])
```

else is not a pattern variable;  
it's matched literally

# Second attempt

```
(switch 3
  [0 "zero"]
  [1 "one"]
  [2 "two"])
```

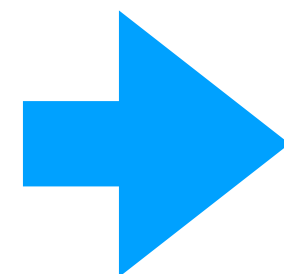
Result is void



```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"]
        [else (void)]))
```

```
(switch 3
  [0 "zero"]
  [1 "one"]
  [2 "two"]
  [else "blah"])
```

Result is "blah"



```
(let ([result 3])
  (cond [(equal? result 0) "zero"]
        [(equal? result 1) "one"]
        [(equal? result 2) "two"]
        [else "blah"]))
```



# Macros match arguments, not evaluate

When a macro is being evaluated, the arguments are matched against the pattern but they aren't evaluated

```
(switch 1
  [0 (displayln "zero")]
  [1 (displayln "one")]
  [2 (displayln "two")]
  [else (displayln "something else")])
```

This prints one

If the arguments were evaluated (well, it'd be an error because 0 isn't a procedure) but it'd also print out zero, one, two, something else

# Hygienic macros

What is printed by the following C code. f is a macro.

```
#include <stdio.h>
```

```
#define f(x) \
do { \
    int y = 10; \
    int z = (x); \
    printf("y=%d z=%d\n", y, z); \
} while (0)
```

```
int main() {
    int y = 5;
    f(y + 2);
    return 0;
}
```

A. y=5 z=7

B. y=5 z=12

C. y=10 z=7

D. y=5 z=12

E. y=10 z=12

# C's macros are "unhygienic"

We can run the code through C's preprocessor which expands macros to see the problem (line breaks added):

```
int main() {  
    int y = 5;  
    do {  
        int y ← = 10;  
        int z = (y + 2);  
        printf("y=%d z=%d\n", y, z);  
    } while (0);  
    return 0;  
}
```

# Scheme/Racket's macros are hygienic

Same macro as before, but in Racket

```
(define-syntax f
  (syntax-rules ()
    [ ( _ x)
      (let* ([y 10]
              [z x])
        (printf "y=~s z=~s\n" y z)) ]))
```

```
(let ([y 5])
  (f (+ y 2)))
```

Prints: y=10 z=7

# Hygienic macros

Unhygienic macros: Macros can introduce variables that shadow variables used in the arguments

- E.g., C's macros are unhygienic

Hygienic macros: Expansion of macros cannot accidentally capture variables

- E.g., Racket's and Rust's macros are hygienic

```
(define-syntax debug-value
  (syntax-rules ()
    [(_ arg)
     (let ([value arg])
       (printf " ~s=~s\n" 'arg value)
       value)]))
(define (f x)
  (* 2 (debug-value x)))
(f 10)
```

What is printed by this code; what is the value of the `(f 10)`?

A. printed: arg=10  
value: 10

B. printed: x=10  
value: 20

C. printed: x=10  
value: 10

D. printed: x=10  
value: 20

# A debug macro

We can use `debug-value` to write a debug macro that wraps a procedure call and prints out all of its arguments:

```
(let ([x 10]
      [y 20]
      [z 30])
  (debug (+ (add1 x) (sub1 y) (* z z))))
```

Prints:

```
(+ (add1 x) (sub1 y) (* z z))
(add1 x)=11
(sub1 y)=19
(* z z)=900
```

Returns: 930



# debug implementation

```
(define-syntax debug
  (syntax-rules ()
    [(_ (f arg ...))
     (begin
       (displayln '(f arg ...))
       (f (debug-value arg) ...))]))
```