

# CS 152: *Programming Language Paradigms*



## Macros

Prof. Tom Austin

San José State University

# Creating control structures with Lambdas (in class)

# Redefining if expressions

```
(define (my-if c thn els)
  (cond
    [(and (list? c) (empty? c)) els]
    [(and (number? c) (= 0 c)) els]
    [(and (boolean? c) (not c)) els]
    [else thn]))
```

# Redefining if expressions

```
(my-if #t 1 0)    ;; returns 1  
(my-if 1 1 0)    ;; also returns 1  
(my-if #f 1 0)   ;; returns 0  
(my-if '() 1 0)  ;; also returns 0
```

```
(my-if #t  
  (displayln "true")  
  (displayln "false"))
```

Why didn't this approach work?

Scheme uses *eager evaluation*.

- Arguments are evaluated first
- Function body is evaluated second
- In our example, we need to evaluate arguments *lazily*
  - that is, only when they are needed

*Macros* allow us to change the behavior of our language as we need.

# What is a macro?

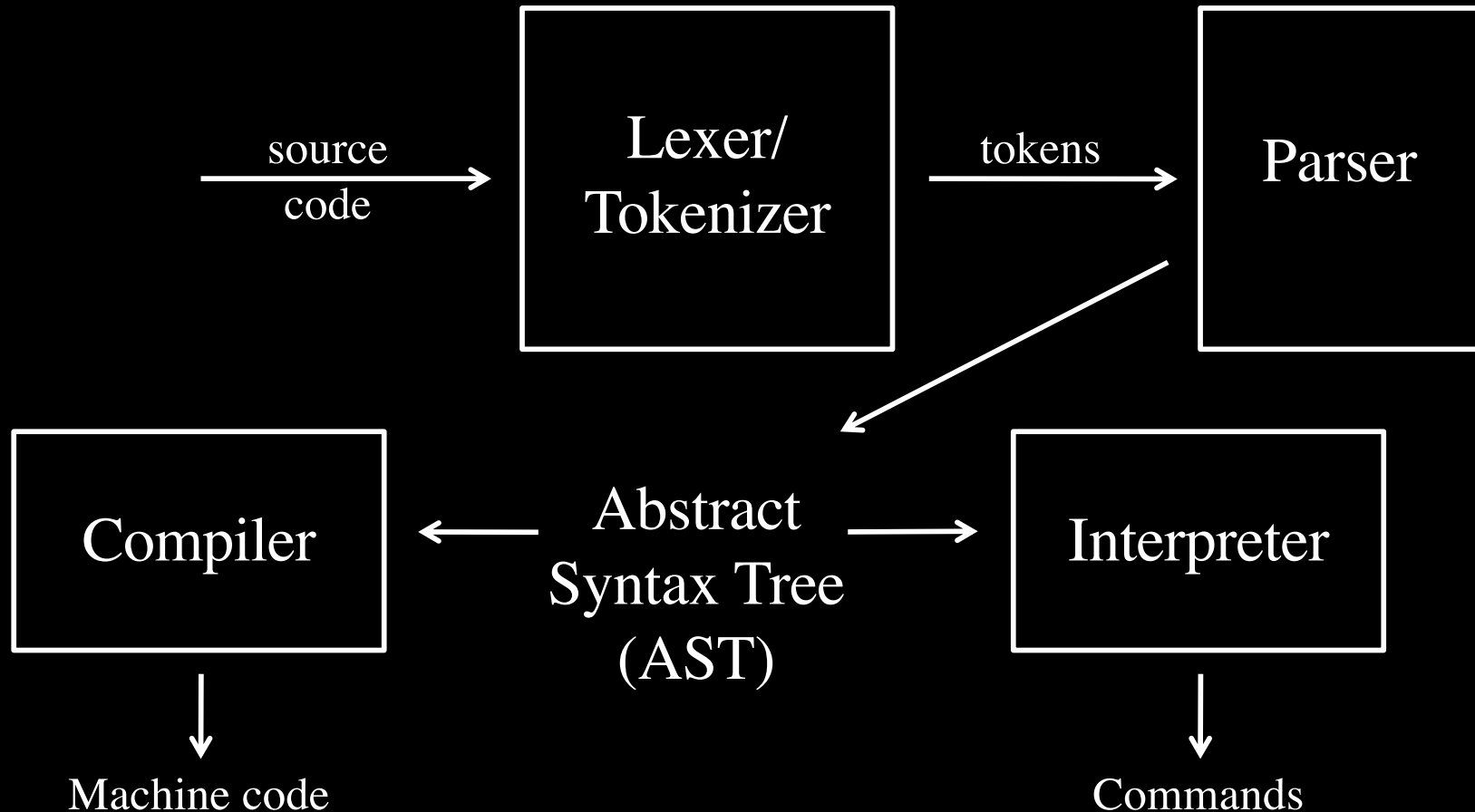
- *macroinstruction*.
- A rule or pattern that specifies how an *input sequence* should be mapped to a *replacement sequence*.



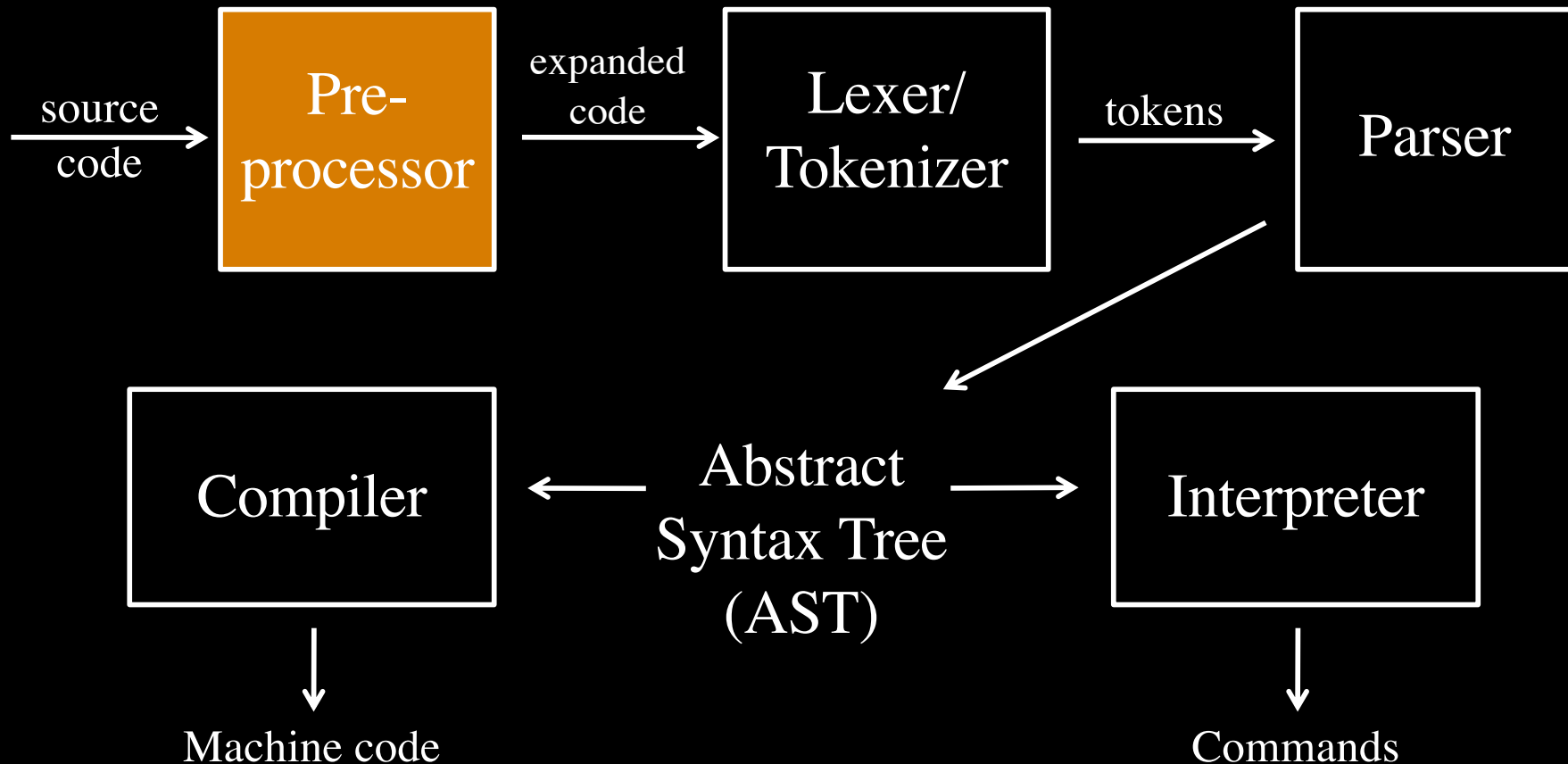
# Text Substitution Macros

- Work by *expanding text*.
- Fast, but limited power.
- Example:
  - C preprocessor

# A Review of Compilers



Some variants work at the token level, but the concept is the same.



# Writing swap in C

(in class)

# C preprocessor example

```
#define PI 3.14159
```

```
#define SWAP(a,b) {int tmp=a;a=b;b=tmp;}
```

```
int main(void) {  
    int x=4, y=5, diam=7, circum=diam*PI;  
    SWAP(x,y);  
}
```

```
int main(void) {  
    int x=4, y=5, diam=7, circum=diam*PI;  
    SWAP(x,y);  
}
```

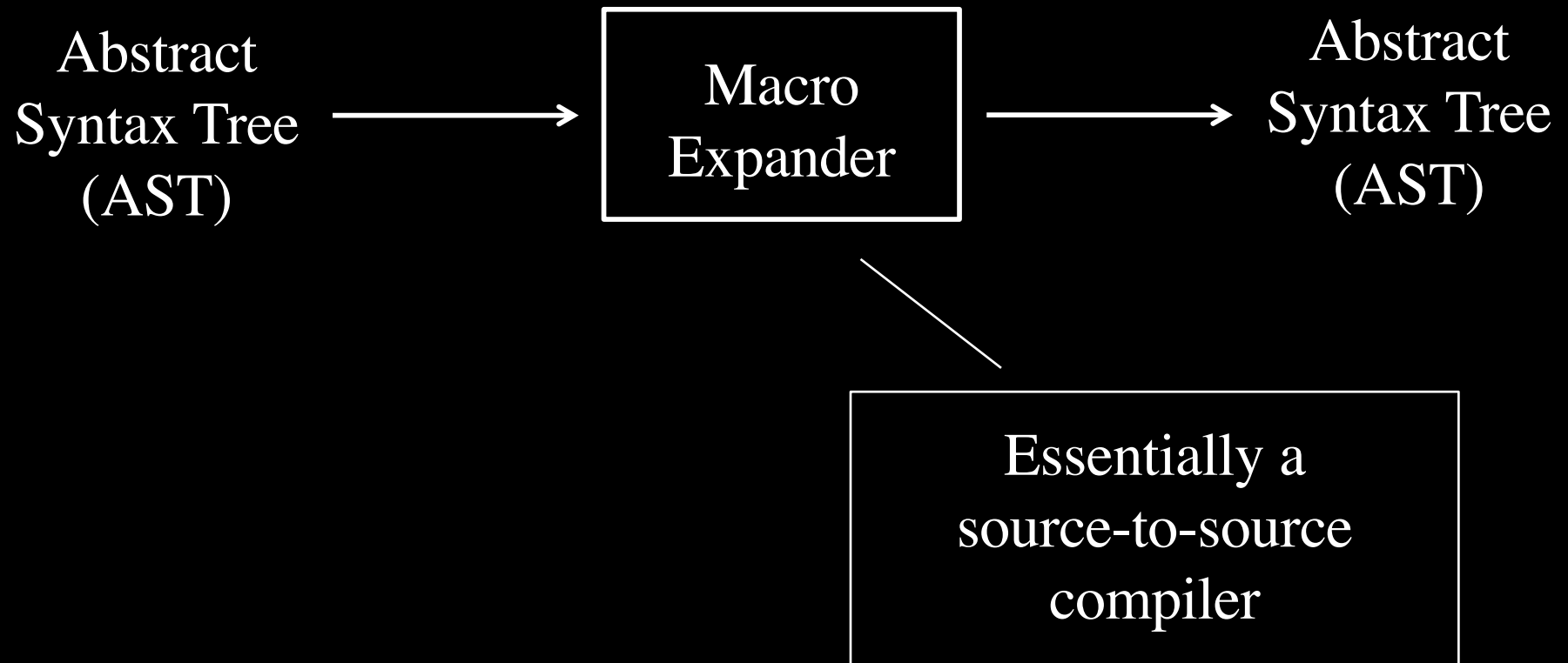


```
int main(void) {  
    int x=4, y=5, diam=7,  
        circum=diam*3.14159;  
    {int tmp=x;x=y;y=tmp;};  
}
```

## Syntactic macros

- Work on abstract syntax trees
- From the Lisp/Scheme family
  - Lisp programs *are* ASTs
- Powerful, but expensive

# Macro expansion process





Many macro systems suffer  
from *inadvertent variable  
capture*.

Let's look at an  
example...

# Accidental Capture Example (in class)

# Hygiene

*Hygienic macros* are macros whose expansion is guaranteed not to cause the *accidental capture of identifiers*.

# Macros in Scheme

- Scheme is noted for its powerful (and hygienic) macro system.
- Is it needed? Aren't lambdas enough?

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

```
(let ([a 7] [b 3])
  (swap a b)
  (displayln a)
  (displayln b))
```

What is the result?

# Pattern Based Macros

- Preserves hygiene
- `define-syntax-rule`
  - matches the given pattern
  - transforms code following the specified template
- `define-syntax`
  - allows multiple patterns
  - supports a variable number of arguments (using the `...` syntax)

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

(let ([a 7] [b 3])
  (swap a b)
  (displayln a)
  (displayln b))
```

What is the result?

## Broken version of my-if

**(define**

  (my-if c thn els)

    (cond [(and (list? c)  
                 (empty? c)) els]

          [(and (number? c)  
                 (= 0 c)) els]

          [(and (boolean? c)  
                 (not c)) els]

    [else thn]))



## Corrected version of my-if

**(define-syntax-rule**

  (my-if c thn els)

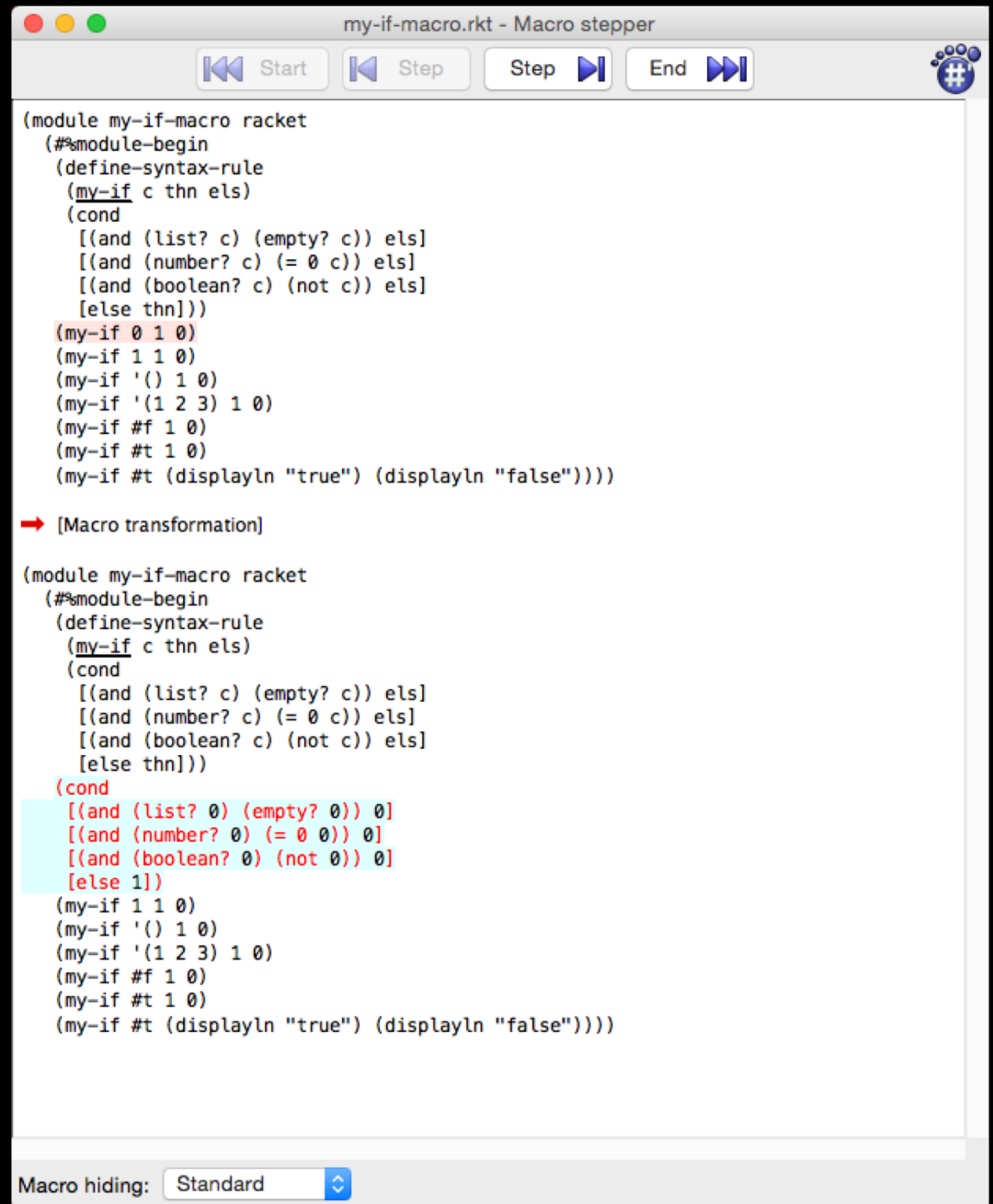
    (cond [(and (list? c)  
                  (empty? c)) els]

          [(and (number? c)  
                  (= 0 c)) els]

          [(and (boolean? c)  
                  (not c)) els]

          [else thn]))

# Using the Macro Stepper in DrRacket



```
my-if-macro.rkt - Macro stepper

(module my-if-macro racket
  (#%module-begin
    (define-syntax-rule
      (my-if c thn els)
      (cond
        [(and (list? c) (empty? c)) els]
        [(and (number? c) (= 0 c)) els]
        [(and (boolean? c) (not c)) els]
        [else thn]))
    (my-if 0 1 0)
    (my-if 1 1 0)
    (my-if '() 1 0)
    (my-if '(1 2 3) 1 0)
    (my-if #f 1 0)
    (my-if #t 1 0)
    (my-if #t (displayln "true") (displayln "false"))))

→ [Macro transformation]

(module my-if-macro racket
  (#%module-begin
    (define-syntax-rule
      (my-if c thn els)
      (cond
        [(and (list? c) (empty? c)) els]
        [(and (number? c) (= 0 c)) els]
        [(and (boolean? c) (not c)) els]
        [else thn]))
    (cond
      [(and (list? 0) (empty? 0)) 0]
      [(and (number? 0) (= 0 0)) 0]
      [(and (boolean? 0) (not 0)) 0]
      [else 1])
    (my-if 1 1 0)
    (my-if '() 1 0)
    (my-if '(1 2 3) 1 0)
    (my-if #f 1 0)
    (my-if #t 1 0)
    (my-if #t (displayln "true") (displayln "false"))))

Macro hiding: Standard
```

# Define-syntax swap function

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

rotate / rotate-all  
(in class)

# Lab

Using `define-syntax`, create a `switch` statement. Sample usage:

```
(define x 99)
(switch x
  [3 (displayln "x is 3")]
  [4 (displayln "x is 4")]
  [5 (displayln "x is 5")]
  [default (displayln
             "none of the above")])
```

For more reading on macros:

- Matthew Flatt, "Pattern-based macros", section 16.1 of "The Racket Guide".  
<http://docs.racket-lang.org/guide/pattern-macros.html>
- Greg Hendershott, "Fear of Macros".  
<http://www.greghendershott.com/fear-of-macros/index.html>