programs as data, quasiquote, macros

lecture 22
announcements

• today's lecture content is in scope for assignments, but not for the final exam
  ◦ will show up on assignments, including the extra credit question on the scheme project :)
• hw05 is due tonight at 11:59pm pt
• vitamins 9 and 10 are due thursday, july 29th, at 8:00am pt
• lab10 is due thursday, july 29th, at 11:59pm pt
• ants early deadline is thursday, july 29th, at 11:59pm pt
• ants project deadline is friday, july 30th, at 11:59pm pt
programs as data

scheme programs consist of expressions, which are either:

- primitive, such as 2, 3.3, #t, +, quotient
- combinations, such as (quotient 10 2), (not #t)

notice that combinations are essentially scheme lists!
programs as data

scheme programs consist of expressions, which are either:

- primitive, such as 2, 3.3, #t, +, quotient
- combinations, such as (quotient 10 2), (not #t)

notice that combinations are essentially scheme lists!

```
scm> (list 'quotient 10 2)
(quotient 10 2)
```
programs as data

scheme programs consist of expressions, which are either:

- primitive, such as $2$, $3.3$, #t, +, quotient
- combinations, such as (quotient 10 2), (not #t)

notice that combinations are essentially scheme lists!

```
scm> (list 'quotient 10 2)  
(quotient 10 2)  
scm> (eval (list 'quotient 10 2))  
5
```

this means that we can write scheme code that writes other scheme code
check for understanding

how would we create a list representing the following expressions?

```
scm>
(print 2)
```
check for understanding

how would we create a list representing the following expressions?

```
scm> (list 'print 2)
(print 2)
scm>
(+ 2 (- 3 2))
```
check for understanding

how would we create a list representing the following expressions?

```scm
scm> (list 'print 2)
(print 2)
scm> (list '+ 2 (list '-' 3 2))
(+ 2 (- 3 2))
scm> (if (> 3 2) (quote greater) (quote smaller))
```
check for understanding

how would we create a list representing the following expressions?

```
scm> (list 'print 2)
(print 2)
scm> (list '+ 2 (list '- 3 2))
(+ 2 (- 3 2))
scm> (list 'if (list '>' 3 2) 'greater 'smaller)
(if (> 3 2) (quote greater) (quote smaller))
```
quasiquotation

very similar to regular quotation, but you can now unquote parts of an expression
quasiquotation

very similar to regular quotation, but you can now unquote parts of an expression

```scheme
scm> '(a b)
(a b)
scm> `(a b)
(a b)
```
quasiquotation

very similar to regular quotation, but you can now unquote parts of an expression

```
 scm> '(a b)
  (a b)
 scm> `'(a b)
  (a b)
 scm> (define b 4)
  b
 scm> '(a ,(+ b 1))
  (a (unquote (+ b 1))
 scm> `'(a ,(+ b 1))
  (a 5)
```
check for understanding

what would scheme display?

```
scm> (define x 5)
x
scm> (define y 10)
y
scm> `(x y)
```

```
1 3
```
check for understanding

what would scheme display?

```
scm> (define x 5)
x
scm> (define y 10)
y
scm> `(x y)
(x y)
scm> `(+ ,x ,y)
```

1 4
check for understanding

what would scheme display?

```
scm> (define x 5)
  x
scm> (define y 10)
  y
scm> `(x y)
  (x y)
scm> `(+ ,x ,y)
  (+ 5 10)
scm> `(if (> ,x ,y) #t #f)
```

1 5
check for understanding

what would scheme display?

```
scm> (define x 5)
x
scm> (define y 10)
y
scm> `(x y)
(x y)
scm> `(+ ,x ,y)
(+ 5 10)
scm> `(if (> ,x ,y) #t #f)
(if (> 5 10) #t #f)
scm> (eval `(if (< ,x ,y) 'success 'sadness))
```
check for understanding

what would scheme display?

```
scm> (define x 5)
x
scm> (define y 10)
y
scm> `(x y)
(x y)
scm> `(+ ,x ,y)
(+ 5 10)
scm> `(if (> ,x ,y) #t #f)
(if (> 5 10) #t #f)
scm> (eval `(if (< ,x ,y) 'success 'sadness))
success
```
generating code

remember `make_adder`?

```python
>>> def make_adder(n):
...    return lambda d: d + n
...

>>> add_2 = make_adder(2)
```

here, calling `add_2` results in python looking up `n` in the `make_adder` frame each time.
generating code

remember `make_adder`?

```python
>>> def make_adder(n):
    ...
    return lambda d: d + n
    ...

>>> add_2 = make_adder(2)
```

Here, calling `add_2` results in Python looking up `n` in the `make_adder` frame each time.

```scheme
scm> (define (make-adder n) `(lambda (d) (+ d ,n)))
make-adder
scm> (eval (make-adder 2))
(lambda (d) (+ d 2))
```

Here, the result of `make-adder` doesn't contain any references to `n`, so we don't need to refer to the `make-adder` frame again. In fact, `make-adder` only returns a list, so it's not the parent of the lambda!
while statements?

what is the sum of the squares of even numbers less than 10, starting at 2?

in python, we can use while loops for this:

```python
x, total = 2, 0
while x < 10:
    total = total + x * x
    x = x + 2
```
**while statements?**

what is the sum of the squares of even numbers less than 10, starting at 2?

in python, we can use while loops for this:

```python
x, total = 2, 0
while x < 10:
    total = total + x * x
    x = x + 2
```

in scheme, we don't have while loops, so we must do this recursively. let's see this in python first.

```python
def f(x, total):
    if x < 10:
        return f(x + 2, total + x * x)
    return total
f(2, 0)
```
what is the sum of the squares of even numbers less than 10, starting at 2?

in python:

```python
def f(x, total):
    if x < 10:
        return f(x + 2, total + x * x)
    return total
f(2, 0)
```

in scheme:

```scheme
(begin
  (define (f x total)
    (if (< x 10)
      (f (+ x 2) (+ total (* x x)))
      total))
  (f 2 0))
```
what is the sum of numbers with squares less than 50, starting at 1?

in python:

```python
def f(x, total):
    if x * x < 50:
        return f(x + 1, total + x)
    return total
f(1, 0)
```

in scheme:

```scheme
(begin
    (define (f x total)
        (if (< (* x x) 50)
            (f (+ x 1) (+ total x))
            total))
    (f 1 0))
```
while statements?

let's see those two side by side.

in python:

```python
def f(x, total):
    if x < 10:
        return f(x + 2, total + x * x)
    return total
f(2, 0)
```

in scheme:

```scheme
(begin
    (define (f x total)
        (if (< x 10)
            (f (+ x 2) (+ total (* x x)))
            total))
    (f 2 0))
```

```python
def f(x, total):
    if x * x < 50:
        return f(x + 1, total + x)
    return total
f(1, 0)
```

```scheme
(begin
    (define (f x total)
        (if (< (* x x) 50)
            (f (+ x 1) (+ total x))
            total))
    (f 1 0))
```
generically, what is the sum of $\text{expr}$ of every $\text{nxt}$ numbers where $\text{condn}$, starting at $\text{init}$?

in python:

```python
def f(x, total):
    if condn(x):
        return f(nxt(x), total + expr(x))
    return total
f(init, 0)
```

in scheme:

```
(begin
  (define (f x total)
    (if (condn x)
        (f (nxt x) (+ total (expr x)))
        total))
  (f init 0))
```
while statements?

what is the sum of $expr$ of every $nxt$ numbers where $condn$, starting at $init$?

let's wrap this in a procedure called $sum\text{-}while$, which takes in the appropriate parameters:

```
(define (sum-while init condn expr nxt)
  (begin
    (define (f x total)
      (if (condn x)
        (f (nxt x) (+ total (expr x)))
        total))
    (f init 0)))
```

```
scm> (sum-while 2 (lambda (x) (< x 10)) (lambda (x) (* x x)) (lambda (x) (+ x 2)))
120
scm> (sum-while 1 (lambda (x) (< (* x x) 50)) (lambda (x) x) (lambda (x) (+ x 1)))
28
```
while statements?

what is the sum of $expr$ of every $nxt$ numbers where $condn$, starting at $init$?

let's use quasiquotation and unquotes to our advantage:

```
(define (sum-while init condn expr nxt)
  `(begin
    (define (f x total)
      (if ,condn
        (f ,nxt (+ total ,expr))
        total))
    (f ,init 0)))
```

```
scm> (eval (sum-while 2 '(< x 10) '(* x x) '(+ x 2)))
120
scm> (eval (sum-while 1 '(< (* x x) 50) 'x '(+ x 1)))
28
```
In Python, we can't add new expressions or statement types. In Scheme, so far, everything has either been a built-in special form or a user-defined procedure. Macros allow us to write our own special forms!

A macro is an operation performed on code before evaluation. Macros exist in many languages, but they're easiest to define correctly in a language like Lisp.

The following code doesn't quite do what we want:

```
scm> (define (twice expr) (list 'begin expr expr))
twice
scm> (twice (print 2))
2
(begin undefined undefined)
```
rules of evaluation

when evaluating procedures, we:

1. evaluate the operator sub-expression
2. evaluate all of the operands
3. apply the procedure on the evaluated operands

scm> (define (twice expr) (list 'begin expr expr))
twice
scm> (twice (print 2))
2
(begin undefined undefined)
rules of evaluation

when evaluating macros, we:

1. evaluate the operator sub-expression
2. call the macro on operands without evaluating the operands
3. evaluate the expression returned by the macro

```
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
scm> (twice (print 2))
2
 2
```
macros without macros

it's possible to replicate macro functionality without macros, but much harder

with macros:

```scheme
scm> (define-macro (twice expr) (list 'begin expr expr))
twice
scm> (twice (print 2))
2
2
```

without macros:

```scheme
scm> (define (twice expr) (list 'begin expr expr))
twice
scm> (eval (twice '(print 2)))
2
2
```
checking truthiness

say we want to check if something's truthy or falsey

```
scm> (define (check val) (if val 'passed 'failed))
check
scm> (define x -2)
x
scm> (check (> x 0))
failed
```
checking truthiness

say we want to check if something's truthy or falsey

```
(scm> (define (check expr) `(if ,expr 'passed '(failed: ,expr)))
check
(scm> (define x -2)
x
(scm> (eval (check '(> x 0))))
(failed: (> x 0))
```
checking truthiness

say we want to check if something's truthy or falsey

```
scm> (define-macro (check expr) `(if ,expr 'passed '(failed: ,expr)))
check
scm> (define x -2)
x
scm> (check (> x 0))
(failed: (> x 0))
```
for macro?

Scheme doesn't have `for` loops... yet. We want to be able to say things like:

```scheme
(scm> (for x '(2 3 4 5) (* x x))
(4 9 16 25))
```

First, let's see how to map items in a list `vals` using some function `fn`.

```scheme
(define (map fn vals)
  (if (null? vals) ()
    (cons (fn (car vals))
      (map fn (cdr vals))))
)
```
for macro?

```
(define (map fn vals)
  (if (null? vals) ()
      (cons (fn (car vals))
            (map fn (cdr vals))))
)
```

we can now say things like `(map (lambda (x) (* x x)) '(2 3 4 5))`, but that's more work than we should have to do. why do we need to explicitly write `lambda`?
for macro?

(define (map fn vals)
  (if (null? vals) ()
    (cons (fn (car vals))
      (map fn (cdr vals)))
  ))

we can now say things like (map (lambda (x) (* x x)) '(2 3 4 5)), but that's more work than we should have to do. why do we need to explicitly write lambda?

(define-macro (for var vals expr)
  `(map (lambda ,var ,expr) ,vals) )

```
office hours are on wednesday, 3:00-5:00 pm
submit anonymous feedback at imvs.me/t/anon
thanks for stopping by :)}