adding your own features to scheme

macros

vanshaj singhania
cs198-175 fa21
cs61a.org/extra

vanshaj [at] berkeley [dot] edu

recall: 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)`

combinations are essentially lists, meaning we can write code that writes code

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

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

recall: programs as data

the following are all valid scheme code, but how do we make it easier to turn this into a template of sorts, in order to be able to reuse it?

```
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))
```

recall: quasiquotation

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

```
scm> `(a b)
(a b)

scm> (define b 4)
b

scm> `(a ,(+ b 1))
(a 5)
```

recall: quasiquotation

we can use this to generate scheme code in a templated form:

```
scm> (define x 5)
x

scm> (define y 10)
y

scm> `(+ x y)
(+ x y)

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

scm> (eval `(if (< ,x ,y) 'success 'not-success))
success
```

generating code

remember `make_adder`?

```
>>> 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`?

```
>>> 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.

```
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!

macros

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
```

how is this different from regular procedures? your macros defines when an operand should be evaluated, not scheme itself! you can delay evaluation as long as you want to -- custom special forms!

macros without macros

it's possible to replicate macro functionality without macros, but much less clean to use

with macros:

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

without macros:

```
scm> (define (twice expr) (list 'begin expr expr))
twice
scm> (eval (twice '(print 2)))
2
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:

```
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:

```
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.

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

`while` statements?

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

in python:

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

in scheme:

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

`while` statements?

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:

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

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

in scheme:

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

```
(begin
    (define (f x total)
        (if (< (* x x) 50)
            (f (+ x 1) (+ total x))
        total))
    (f 1 0))
```

`while` statements?

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

in 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` is true, starting at `init`?

let's wrap this in a procedure called `sum-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` is true, starting at `init`?

let's use quasiquotation and unquotes to our advantage to make this less repetitive:

```scheme
(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
```

`while` statements?

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

here's the same code as before, but turned into a macro:

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

```
scm> (sum-while 2 (< x 10) (* x x) (+ x 2)) ; no eval, no quoting
120
scm> (sum-while 1 (< (* x x) 50) x (+ x 1)) ; much cleaner to read, isn't it?
28
```

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
```

can't really check ***what's*** failing, as the `check` procedure only receives the ***evaluated result*** of `val`!

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:

```
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`.

```
(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)
)
```

past exam problem: fa19 final q9 (macro lens)

implement `partial`, a macro that takes a call expression that is missing its last operand. a call to `partial` evaluates to a one-argument procedure that takes a value `y` and returns the result of evaluating `call` extended to include an additional operand `y` at the end.

```
;; a macro that creates a procedure from a partial call expression missing the last operand.
;; (define add-two (partial (+ 1 1))) -> (lambda (y) (+ 1 1 y))
;; (add-two 3) -> 5 by evaluating (+ 1 1 3)
;;
;; (define eq-5 (partial (equal? (+ 2 3)))) -> (lambda (y) (equal? (+ 2 3) y))
;; (eq-5 (+ 3 2)) -> #t by evaluating (equal? (+ 2 3) 5)
;;
;; ((partial (append '(1 2))) '(3 4)) -> (1 2 3 4)
(define-macro (partial call)



)
```

past exam problem: fa19 final q9 (macro lens)

implement `partial`, a macro that takes a call expression that is missing its last operand. a call to `partial` evaluates to a one-argument procedure that takes a value `y` and returns the result of evaluating `call` extended to include an additional operand `y` at the end.

```scheme
;; a macro that creates a procedure from a partial call expression missing the last operand.
;; (define add-two (partial (+ 1 1))) -> (lambda (y) (+ 1 1 y))
;; (add-two 3) -> 5 by evaluating (+ 1 1 3)
;;
;; (define eq-5 (partial (equal? (+ 2 3)))) -> (lambda (y) (equal? (+ 2 3) y))
;; (eq-5 (+ 3 2)) -> #t by evaluating (equal? (+ 2 3) 5)
;;
;; ((partial (append '(1 2))) '(3 4)) -> (1 2 3 4)
(define-macro (partial call)
    `(lambda (y) ,(append call (list 'y)))

)
```

past exam problem: sp19 final q8

the `if` special form has been removed from scheme. implement an `if`-macro using only `and`, `or`, and `not`.

```
(define-macro (if condition then else)


)
```

past exam problem: sp19 final q8

the `if` special form has been removed from scheme. implement an `if`-macro using only `and`, `or`, and `not`.

```
(define-macro (if condition then else)
    `(or (and ,condition ,then) ,else)

)
```

submit anonymous feedback at imvs.me/t/anon

thanks for stopping by :)

vanshaj [at] berkeley [dot] edu