Adding Custom Special Forms to Scheme

Macros

CS61A Spring 2022

MWF 2:00-3:00pm

cs61a.org

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> (begin (define x 5) (define y 10))

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`?

```
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.

Macros

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
```

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

How is this different from regular procedures? Your macros, not Scheme, define when an operand should be evaluated. You can delay evaluation as much as you want to -- custom special forms!

Macros Without Macros

It's possible to replicate macro functionality without macros, but much less clean:

```
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?

Scheme doesn't have while loops, so we need recursion. In Python:

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

`while` Statements?

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

In Scheme:

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

`while` Statements?

What is the sum of numbers with squares less than 50, starting at 1?

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

(begin
    (define (f x tot)
        (if (< (* x x) 50)
            (f (+ x 1) (+ tot x))
            tot))
    (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:

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

`while` Statements?

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

In Scheme:

```
(begin
    (define (f x tot)
        (if (condn x)
            (f (nxt x) (+ tot (expr x)))
            tot))
    (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)))
```

`while` Statements?

We might use the `sum-while` procedure as follows:

```
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:

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

while Statements?

We might use our new `sum-while` procedure as follows:

```
scm> (eval (sum-while 2 '(< x 10) '(* x x) '(+ x 2)))
120
scm> (eval (sum-while 1 '(< (* x x) 50) 'x '(+ x 1)))
28
```

Much cleaner already!

`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)))
```

`while` Statements?

We might use our shiny new `sum-while` macro as follows:

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

No `eval`, no quoting. Much nicer to read, isn't it?

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))
```

`def` Statements

We want to write a macro that simulates Python's `def` statements in Scheme, in order to say things like `(def f(x y) (+ x y))`.

```
scm> (define-macro (def n args b) `(define ,n (lambda ,args ,b)))
def

scm> (def f(x y) (+ x y))
f

scm> (f 5 2)
7
```

Thunk Macro

We want to write a macro that takes in some `expr` and turns it into a Thunk. Remember that a Thunk is a no-argument lambda function that, when called, evaluates and returns the `expr` it contains.

```
scm> (define-macro (thunkify expr) `(lambda () ,expr))
thunkify

scm> (define thunk (thunkify (+ 2 4)))
thunk

scm> (thunk)
6
```

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

`for` Macro?

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

We can now say things like `(map-fn (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?

We can now say things like `(map-fn (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-fn (lambda (,var) ,expr) ,vals)
)

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

Success!

Submit anonymous feedback at imvs.me/t/anon

Thanks for stopping by :)

vanshaj [at] berkeley [dot] edu