CS61A Lecture 30

Friday, November 8, 2019

Announcements

- Homework 8 is due Thursday, 11/14.
- Cats composition revsisions due Tuesday, 11/12.
- Scheme project is due Wednesday, 11/20.

Programs as Data

Scheme programs consist of expressions, which can be primitive or combinations.

The built-in Scheme list data structure (which is a linked list) represents linked lists.

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

In such a language, we can write programs that writes other programs.

```
> (+ 1 2)
3
> (list + 1 2)
(#[+] 1 2)
```

The problem with this is that `#[+]` is not how Scheme represents the operators. This is why we use quotation: we want Scheme to not evaluate the expression yet.

```
> (list '+ 1 2)
(+ 1 2)
> (list '+ 1 (+ 2 3))
(+ 1 5)
```

So, using this principle, we could theoretically write Scheme programs that return other programs. Previously, we had programs that return values:

```
> (define (fact n) (if (= n 0) 1 (* n (fact (-n 1)))))
> (fact 5)
120
```

But there's nothing that stops us from returning other programs:

```
> (define (fact-exp n) (if (= n 0) 1 (list '* n (fact-exp (- n 1)))))
> (fact-exp 5)
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
```

This is a perfectly valid program that returns an unevaluated expression.

```
> (eval (fact-exp 5))
120
```

Here's another example of the same style:

```
> (define (fib n) (if (<= n 1) n (+ (fib-exp (- n 2)) (fib-exp (- n 1)))))
> (define (fib-exp n) (if (<= n 1) n (list '+ (fib-exp (- n 2)) (fib-exp (- n 1)))))
> (fib 6)
8
> (fib-exp 6)
(+ (+ (+ 0 1) (+ 1 (+ 0 1))) (+ (+ 1 (+ 0 1)) (+ (+ 0 1) (+ 1 (+ 0 1)))))
> (eval (fib-exp 6))
8
```

Macros

Macros are a feature of Scheme that can be performed on the source code of a program before evaluation. Macros exist in many languages, but are easiest to define correctly in a language like Scheme.

In Scheme, we've so far seen special forms like `if`, `cond`, and `begin`, which are one of a small inventory. Everything else is a procedure. What a mcro enables you to do is extend that inventory of special forms, defining ways in which the language is evaluated.

The way we do this is that we describe the ways to take parts of a special form, construct a regular piece of Scheme code out of them, and that evaluate that piece of code.

Scheme has a special form, `define-macro`, that defines a source code transformation.

```
(define-macro (twice expr)
    (list 'begin expr expr)
)
```

This is the evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first.
- Evaluate the expression returned from the macro procedure.

The above code is an example of why macros are useful. If we tried doing it without macros:

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

It only prints 2 once. Why? Because it evaluates `print 2` as it is called in the call expression, not in the context of the body.

With the macro (from the code in the example above):

```
> (twice (print 2))
2
2
```

Here's another example. Let's say I want to define a procedure that checks if a value is true or false:

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

Wouldn't it be nice if we could know what failed the check? Not just the value, but the expression itself? Without a macro, the expression is evaluated when the procedure is called, and that's it. You've lost access to it. The only way to do that would be to consciously pass in the expression quoted.

But with macros...

Well, first we can do precisely the same thing we did before:

```
(define-macro (check-macro expr)
    (list 'if expr ''passed ''failed)
)
```

It's double quoted to say that the result of evaluating the if clause should be the quoted expression `'passed`.

```
(define-macro (check-macro expr)
    (list 'if expr ''passed (list 'quote (list 'failed: expr)))
)
```

And now, we have access to the expression within the body of the procedure:

```
> (define x -2)
x
> (check (> x 0))
(failed: (> x 0))
```

For Statement

Scheme doesn't have a for statement, but we can create one ourself with a macro.

To define a macro that evaluates an expression for each value in a sequence:

```
> (for x '(2 3 4 5) (* x x))
```

```
(4 9 16 25)
```

Well first, we have to define the map function:

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

We can do most of a for statement with this `map` function:

```
> (map (lambda (x) x* x) '(2 3 4 5))
(4 9 16 25)
```

Now, to finish off our `for` statement, our macro should build the `map` expression from before:

```
(define-macro (for sym vals expr)
    (list 'map (list 'lambda (list sym) expr) vals)
)
```

This `for` statement works now! Excellent. We've mixed together something that gets evaluated (the list), something that gets evaluated for every element (the expression) and something that never gets evaluated (the symbol).

Quasi-Quote Special Form

The quasi-quote special form is similar to the quote special form. The expression does not get evaluated, instead the value is the expression itself. What's special about quasi-quotation is that parts of that expression can be evaluated.

```
> (define b 2)
2
> '( a b c)
(a b c)
```

The fact that b is 2 doesn't matter. We're just writing down a list of symbols. The same is true when you quasi-quote:

```
> `(a b c)
(a b c)
```

The power of quasi-quote is that you can selectively choose parts of the expression to be unquoted:

```
> `(a ,b c)
(a 2 c)
```

You can also unquote entire expressions:

```
> `(a ,(+ b 5) c)
(a 7 c)
```

And if you unquote something that can't be evaluated, you will get an error.

You can only unquote if the expression is quasi-quoted. If you tried to unquote a regular quoted expression:

```
> '(a ,b c)
(a (unquote b) c)
```

Macros are the reason quasi-quoting exists in the language. It makes it much more convenient to write them:

```
> (define expr (* x x))
expr
> `(lambda (x) ,expr)
(lambda (x) (* x x))
```

Watch how quasi-quoting can simplify our `check` function earlier.

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

The comma is the failed clause is a little confusing. It unquotes the outer quasi-quote, so it correctly evaluates `expr` to the expression it is bound to, but it does not get evaluated to a single value because it is within a quote. Without it:

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

Hopefully, quasi-quoting makes macros more comprehensible to you.