

# 6.001 Tutorial 12

David Ziegler

November 29, 2004

## 1 Administrivia

- Project 5 due Friday, and it's short!
- Only one more tutorial! Send me suggestions: david@ziegler.ws.

## 2 unless

One of the most fundamental types of expression in any language is conditionals – a way of saying “do this, if something is the case.” We’ve got one type of conditional in Scheme, the `if` expression. A number of other languages have the converse, `unless`. For example, in Perl you can write

```
print "Uh oh!"  
  unless (everything eq "alright");
```

This means just what it looks like; print a message unless something is true. We’re going to add the `unless` expression type to Scheme, with a different syntax:

```
(unless test  
  consequent)
```

The advantage of this? If we want to do something if a condition is not true, now we don’t have to have an empty consequent in an `if`.

Assuming we have `unless-test` and `unless-consequent` available...

```
(define (eval-unless exp env)
```

```
(define (desugar-unless exp)
```

## 3 and

One of the problems from project 5 is to handle the `or` special form. Both `and` and `or` have to be handled as special forms, because they are “short circuiting.” When you evaluate an `and` special form, you evaluate each expression from left to right, and as soon as you get back `#f`, you stop, and don’t evaluate any of the other expressions. Similarly, with an `or` expression, you stop as soon as you get something that is true.

The precise rule for `and`: evaluate each expression from left to right. The value of the first expression which evaluates to `#f` is returned. Any remaining expressions are not evaluated. If all expressions evaluate to true values, the value of the last expression is returned. If there are no expressions, then `#t` is returned.

Assuming we have `and-exprs`, which gives us the list of expressions...

```
(define (eval-and exp env)
```

```
(define (desugar-and exp)
```

## 4 do

The `do` special form is a way of expressing many types of loops in Scheme (for those of you who can't stand recursion).

```
(do ((var init step)
    ...)
    (test expr ...)
    cmd
    ...)
```

```
(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
      (sum 0 (+ sum (car x))))
      ((null? x) sum)))
```

```
(define (do-vars exp)
```

```
(define (do-inits exp)
```

```
(define (do-steps exp)
```

```
(define (do-test exp)
```

```
(define (do-exprs exp)
```

```
(define (do-cmds exp)
```

```
(define (eval-do exp env)
```

```
(define (desugar-do exp)
```