Class 3: More on evaluation

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011

Scheme is lists!

Everything in Scheme that looks like a list is a list.
You have been using lists, but usually asking Scheme to **evaluate** them.

Scheme evaluates a list by using a general rule:

- First, turn a list of expressions `(e1 e2 e3 ...)` into a list of atoms `(a1 a2 a3 ...)` by recursively evaluating each `e1`, `e2`, etc.
- Then, apply the procedure `a1` to the arguments `a2`, `a3`, ...

Anything that is *not* a list (i.e., an atom) just evaluates to itself.

Special Forms

The only exceptions to the evaluation rule are the **special forms**.

Special forms we have seen: `define`, `if`, `cond`, `and`, `or`.

What makes these "special" is that they
do not (always) evaluate (all) their arguments.

Example: evaluating `(5)` gives an error, but
`(if #f (5) 6)` just returns 6 — it never evaluates the "`(5)`" part.

Scheme evaluation and unevaluation

We can use the built-in function `eval` to evaluate a Scheme expression within Scheme!

- Try `(eval (list + 1 2))`

We can also ask Scheme **not** to evaluate an expression by using the special form `quote`.

- Try `(quote (+ 1 2))`

Quoting

There is a convenient shortcut of `quote`: Putting an apostrophe before the expression-to-be-quoted.
For example, `'(1 2 3)` is the same as `(list 1 2 3)`.

This gives us a synonym for `null`: `'()`.
In fact, `'()` is the preferred Scheme way of writing an empty list.

Creating nested lists also becomes trivial:
`'(1 (2 3) 4)` is equivalent to `(list 1 (list 2 3) 4)`

Symbols

An unevaluated identifier is called a **symbol**.
(Note: the predicate `symbol?` is useful here.)

Symbols are useful beyond evaluation and quoting.
We often use them like ENUMs in C++.
Examples: units, months, grades

Symbols are often used to **tag** data: `(cons 10.3 'feet)`

Some exercises

1. Write a function `sign` that takes a number and returns the symbol `'positive`, `'negative`, or `'zero`, as appropriate.
2. Write a simple quoted expression that is equivalent to `(cons (cons 3 (cons 'q null)) (cons 'a null))`.
3. Write a function that takes a list of numbers and adds them up using the + function. (Hint: first build this expression using `cons`, then evaluate it using `eval`.)
4. Repeat #3 using the built-in `apply` function.

The need for local variables

This code finds the largest number in a list:

```
(define (lmax L)
  (cond [(null? (cdr L)) (car L)]
        [(>= (car L) (lmax (cdr L))) (car L)]
        [else (lmax (cdr L))]))
```


The `let` special form

Scheme provides `let` as a way to re-use temporary values:

```
(define (lmax L)
  (if (null? (cdr L))
      (car L)
      (let ((rest-max (lmax (cdr L))))
        (if (>= (car L) rest-max)
            (car L)
            rest-max))))
```

Note the **extra parentheses** — these allow multiple temporary variables:
`(let ((a 5) (b 6)) (+ a b))` ⇒ 11

More exercises

1. Write a Scheme expression that computes the formula $5x^2y + 3xy - x + 4y$ at the point $(x, y) = (1.5, 2.5)$.

2. Write a Scheme function `(f x y)` that computes the formula $5x^2y + 3xy - x + 4y$ at any given point.

3. Simulate the following Java code as a series of nested `lets`:

```
int x = 1;
x += 3;
x *= 12;
return x;
```