Outline

- Errata
- Homework #2
- Functions
  - review
  - parameter lists
  - anonymous fns
  - closures
  - recursion
  - tail recursion
- Example
Functions

- Defined with defun, but
  - just a convenience
- Real issue
  - symbol-function binding
Calling Functions

- Normal evaluation
  - (+ 1 2 3)

- Others
  - funcall
  - apply
  - avoid eval
Multiple values

- Let equivalent
  - multiple-value-bind
-setq equivalent
  - multiple-value-setq
- Example
  (multiple-value-bind (div rem) (floor num 10)
  ... rest of code ...)
- Also
  - multiple-value-list
Parameter Lists

- **keywords**
  
  (defun foo (a &key b c) ... )
  (foo 5 :b 10 :c 19)
  (foo 5)

- **rest**
  
  (defun foo (&rest lst) ... )
  (foo)
  (foo 10 15 25 30)

- **optional**
  
  (defun foo (a &optional b c) ... )
  (foo 5)
  (foo 5 10 19)
Anonymous Functions

- Like anonymous classes in Java
  - but...
- lambda
  - a macro that creates function objects
    
    (lambda (x) (* x x))

- Can create functions where needed
  - useful as functional arguments
    
    (mapcar #'(lambda (x) (* x x)) '(1 2 3))

- Can write functions to return functional values
Example

- numeric series
- evaluator
Closures

- **Something odd?**
  ```lisp
  (defun evaluator (poly)
    #'(lambda (x) (evaluate-poly poly x))
  )
  ```

- **Closure**
  - binds variables in external scope
  - keeps them when the scope disappears
Environment

- Evaluation takes place in an environment
  - bindings for symbols
- Hierarchy of environments
  - global environment
  - local environments
- "Lexical context"
Example

(defun bar (arg1 arg2)
  (let ((var1 (baz arg1 arg2)))
    #'(lambda () (foo var1))))

(setf fn (bar 'a 'b))
When Closures is Created

- Lambda
  - (lambda ()
    - (foo var1 ...)

- Let
  - (let ((var1...)

- Function
  - (arg1 arg2)

- Global environment
  - defun ...
  - defparameter ...
When Closure is Returned

```
(let ((var1...

(lambda ()
  (foo var1 ...)

(let ((var1...

fn
```
Encapsulation

- Achieves OO goal
  - variables in closure are totally inaccessible
  - no way to refer to their bindings
  - no way to even access that environment
Example

- Sharing a binding
Compare with Java

- Anonymous classes
- External values only if
  - instance variables
  - final variables
Anonymous Class (illegal)

```java
JButton okButton = new JButton("OK");
okButton.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            mCanceled = false;
            okButton.setEnabled(false);
        }
    });
```
Anonymous Class (legal)

```java
final JButton okButton = new JButton ("OK");
okButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        m_canceled = false;
        okButton.setEnabled(false);
    }
});

// BUT
okButton = new JButton ("Yep"); // Illegal
```
Simulating a Closure in Java

```java
final JButton [] arOkButton = new JButton [1];
arOkButton[0] = new JButton ("OK");
arOkButton[0].addActionListener (new ActionListener ()
{
    public void actionPerformed (ActionEvent e)
    {
        m_canceled = false;
arOkButton[0].setEnabled(false);
    }
});

arOkButton[0] = new JButton ("Yep"); // Legal
```
Recursion

- Basic framework
  - solve simplest problem "base case"
  - solve one piece
  - combine with the solution to smaller problem

- Why it works
  - eventually some call = base case
  - then all partial solutions are combined
Recursive Statement

- Some problems have a natural recursive statement
  - maze following
  - pattern matching
Example

- pattern matching
Double Recursion

- **Trees**
  - operating on the "current" element isn't simple
  - recurse on both car and cdr

- **Strategy**
  - base case
  - combine solution-to-car with solution-to-cdr
Example

- count-leaves
Recursive Stack

(defun factorial (n)
  (if (= 0 n) 1 (* n (factorial (1- n)))))

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 0</td>
<td>1</td>
</tr>
<tr>
<td>n = 1</td>
<td>(* 1 1)</td>
</tr>
<tr>
<td>n = 2</td>
<td>(* 2 1)</td>
</tr>
<tr>
<td>n = 3</td>
<td>(* 3 2)</td>
</tr>
<tr>
<td>n = 4</td>
<td>(* 4 6)</td>
</tr>
<tr>
<td>n = 5</td>
<td>(* 5 24)</td>
</tr>
<tr>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>
Tail Recursion

- Avoid revisiting previous context
  - = no after-the-fact assembly

- New formulation
  - accumulate answer in parameter
  - base case = return answer
  - recursive step
    - do partial solution and pass as answer parameter

- Effect
  - do need to revisit context
  - recursive call = GOTO!
Tail Recursive Factorial

(defun factorial (n) (factorial-1 n 1))
(defun factorial-1 (n ans)
  (if (= 0 n) ans (factorial-1 (1- n) (* n ans))))

| n = 0, ans = 120 | 120 |
| n = 1, ans = 120 | 120 |
| n = 2, ans = 60  | 120 |
| n = 3, ans = 20  | 120 |
| n = 4, ans = 5   | 120 |
| n = 5, ans = 1   | 120 |
Example

- average
Symbolic Programming Example

- Cards