Macros

CSC 358/458
5.20.2003
Outline

- Homework #6
- Macro Intro
- Backquote
- Macros
- Modify macros
- Read Macros
Homework #6
### Digression

<table>
<thead>
<tr>
<th>Test</th>
<th>Lisp</th>
<th>Java</th>
<th>Python</th>
<th>Perl</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception handling</td>
<td>0.01</td>
<td>0.90</td>
<td>1.54</td>
<td>1.73</td>
<td>1.00</td>
</tr>
<tr>
<td>hash access</td>
<td>1.06</td>
<td>3.23</td>
<td>4.01</td>
<td>1.85</td>
<td>1.00</td>
</tr>
<tr>
<td>sum numbers from file</td>
<td>7.54</td>
<td>2.63</td>
<td>8.34</td>
<td>2.49</td>
<td>1.00</td>
</tr>
<tr>
<td>reverse lines</td>
<td>1.61</td>
<td>1.22</td>
<td>1.38</td>
<td>1.25</td>
<td>1.00</td>
</tr>
<tr>
<td>matrix multiplication</td>
<td>3.30</td>
<td>8.90</td>
<td>278.00</td>
<td>226.00</td>
<td>1.00</td>
</tr>
<tr>
<td>heapsort</td>
<td>1.67</td>
<td>7.00</td>
<td>84.42</td>
<td>75.67</td>
<td>1.00</td>
</tr>
<tr>
<td>array access</td>
<td>1.75</td>
<td>6.83</td>
<td>141.08</td>
<td>127.25</td>
<td>1.00</td>
</tr>
<tr>
<td>list processing</td>
<td>0.93</td>
<td>20.47</td>
<td>20.33</td>
<td>11.27</td>
<td>1.00</td>
</tr>
<tr>
<td>object instantiation</td>
<td>1.32</td>
<td>2.39</td>
<td>49.11</td>
<td>89.21</td>
<td>1.00</td>
</tr>
<tr>
<td>word count</td>
<td>0.73</td>
<td>4.61</td>
<td>2.57</td>
<td>1.64</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>25% to 75%</strong></td>
<td>0.93 to 1.67</td>
<td>2.63 to 7.00</td>
<td>2.57 to 84.42</td>
<td>1.73 to 89.21</td>
<td>1.00 to 1.00</td>
</tr>
</tbody>
</table>

Relative speeds of 5 languages on 10 benchmarks from [The Great Computer Language Shootout](https://gallium.oxide.ca/). Speeds are normalized so the g++ compiler for C++ is 1.00, so 2.00 means twice as slow; 0.01 means 100 times faster. Background colors are coded according to legend on right. The last line estimates the 25% to 75% quartiles by throwing out the bottom two and top two scores for each language.
Function Evaluation

- Function call
  - \((+ (* 2 3) (/ 15 5))\)

- What happens
  - Evaluate each argument
  - Evaluate the function using the values
What about

- `(let ((a 6) (b 3)) (+ a b))`  
- Clearly not function evaluation!
Macro Expansion

- A different pattern of evaluation
  - The arguments are not evaluated
- The macro is evaluated
- Then the result is evaluated
Macroexpand

- Evaluation
  - does the first part
  - but not the second
- Useful for debugging your macros
Example

- POP
Backquote

- Writing lots of LIST expressions is hard
- Would be better to have a pattern we can fill in
- This is the function of backquote
  
  `
Backquote facility

- Backquote is like quote
  - `(a b) => (a b)

- With substitution
  - `(let ((b 5)) `(a ,b)) => (a 5)
  - `(let ((b 5 6)) `(a ,b)) => (a (5 6))
  - we can use this outside of macros, too

- Also "splicing"
  - `(let ((b '(5 6)) `(a ,@b)) => (a 5 6)
Example

- REPEAT-N
Problems?

- Multiple evaluation
- Variable capture
Problem

- Macro code must work wherever it is expanded
- Can't make assumptions about lexical environment
- Opportunity
  - can make our own lexical environment
Example

- LET
- (LET*)
Macro Applications

- New top-level forms
  - <-
  - ?-
- Destructive operations
Creating structure objects

- Employee
- List of employees created
- Functional version
- Macro version
- WITH- macros
  - intentional capture
  - destructuring
Digression ... Hash Tables

- Hash tables
  - like Java HashMap data structure
- Uses hashing on a key to store a value
Hash Tables

- **Functions**
  - `(make-hash-table :test test)`
  - `(gethash key table)`
  - `(setf (gethash key table) val)`
  - `(maphash #'(lambda (key val) ...) tbl)`

- **Test = #'equal**
  - for strings and lists as keys
  - default #'eql
Indexed version
A macro for indexed structs
Setf macros

- `(employee-full-name e)`
  - "Burke, Robin"

- What if we wanted?
  - `(setf (employee-full-name e) "Burke, Robin")`

- defsetf
  - args must be location, value
  - must return value
Example
Read macros

- #P"C:\temp\foo.txt"
  - somehow this creates a pathname structure

- #S(POINT :X 2 :Y 3)
  - somehow this creates a structure

- #2A((1 2 1) (0 -1 1))
  - somehow creates an array

- (1 2 3)
  - somehow this creates a list
The Reader

- Input is just a stream
  - the reader makes evaluation happen
- Certain characters are special
  - just like LET and DO are special
- Programmers have control over this
Readable

- A translation between characters and actions the reader should take
  - in most cases, something basic
  - in special cases, something different

- Examples
  - A, c, ; , #, (, )
Defining your own

- Generally considered a bad idea
  - to overwrite the reader's behavior on regular characters
- Dispatch character #
  - place for the programmer to add read macros
Read macro functions

- Two arguments
  - stream
  - char
- The function should
  - read from the stream
  - return an appropriate object
Dispatch read macro functions

- Three arguments
  - stream
  - subchar
  - parameter
- \#2A((1 2) (2 1))
  - subchar = A
  - parameter = 2
Example

- Read macro for polynomials
  - \#Y"3x^3+2x-7"
Homework #7

- Read macro for list iterators
  - `#I(1 2 3 4)`
  - returns a list iterator

- with-list-iterator
  - `(with-list-iterator (i (a b c d ef))
    (next) (format t "~A~%" (next)))`

- doiter
  - `(doiter (v #I(a b c))
    (format t "~A~%" v))`