# Introduction to Lisp

Jordan Biondo
Sean Fisk

Grand Valley State University

March 1, 2013

# Introduction



- List Processing, also know as Lisp
- Second oldest currently used high-level programming language
- Invented by John McCarthy at MIT in 1958
- Leading family of functional programming languages

# Functional Programming



- Functional languages view computation as the evaluation of mathematical functions.
- Functional programming is based on lambda calculus.
- Functions have no side effects:
  - They avoid mutable data, i.e., changing values outside of a function's scope.
  - Lisp can be written functionally, but is not a purely functional language. It may also be written with typical imperative or object-oriented approaches.

# Structure of the Language



- Parenthesized prefix notation
- Data is contained in S-expressions
- Code is data
- Everything in Lisp is either an atom or a list.

# Atoms



Represent the most basic data types in Lisp.

Examples:

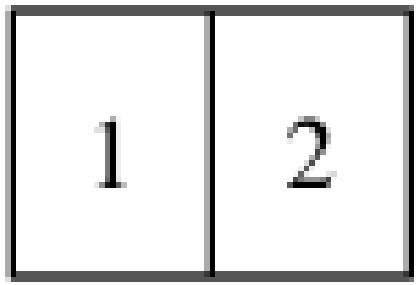
| | |
|---|---|
| Numbers | `9, 12.2, 9e10, \#x2f, 10/3` |
| Strings | `"Bob", "Lisp is awesome"` |
| Characters | `#\a, #\linefeed` |

# Cons Cells



- Stands for "construct"
- Data structure containing two pointers
- Like a linked-list cell with two elements
  - A pointer to the cells value
  - A pointer to the next cell

Creating a cons cell: `(cons 1 2)`



# car and cdr



`car`:

- `car` returns the value of the first element of a cons cell
- `(car (cons 1 2)) -> 1`
- Alternate notation: `(first (cons 1 2))`
- `car` stands for "Contents of the Address part of Register"

`cdr`:

- `cdr` returns the value of the second element of a cons cell
- `(cdr (cons 1 2)) -> 2`
- Alternate notation: `(rest (cons 1 2))`
- `cdr` stands for "Contents of the Decrement part of Register"

The names are historical and do not have any current meaning.

# Lists



- Ordered collection of cons cells.
- The `cdr` of each cons is a pointer to the next cons, just like a linked list.
- The last element in a list has a `nil` `cdr`, signifying the end of the list.
- Nested lists are expressed in a parenthesized notation known as an **S-expression**.
- S-expressions can be though of as trees of cons cells.

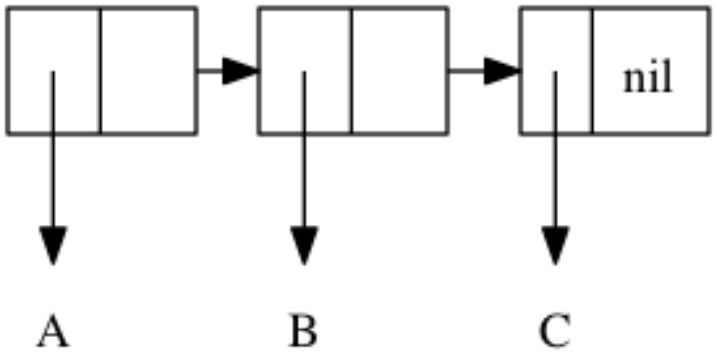
Example:

```
((kurmas wolffe engelsma nandigam)
 (c ruby lisp ada))
```

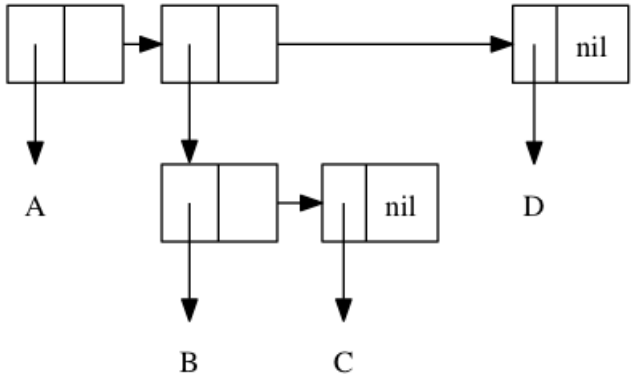
# More List Examples



```
(A B C)
```




```
(A (B C) D)
```



# Lisp Environment



- We will be using **GNU Clisp** as our Common Lisp implementation.
- Log in to your EOS account using your favorite SSH client (e.g., PuTTY on Windows, `ssh` on OS X or GNU/Linux) or through VNC.
- Start an editor and a terminal emulator.
- Run Clisp by typing `clisp` on the command-line. You will see a Common Lisp REPL (Read-Eval-Print-Loop) appear.
- Files can be loaded into the Lisp environment in two ways:
  - Load the file in Clisp by running `(load "myfile.lisp")` in the REPL.
  - Initialize Clisp with the file by running `clisp -i myfile.lisp` on the command-line.

# Editing Lisp



Preferably, use an editor with parentheses matching.

The Emacs Editor

- Emacs is an editor written mostly in Lisp (Emacs Lisp).
- It is very adept at editing Lisp code, especially with `paredit`.
- Preferred editor for Jordon and Sean.

# Editing Lisp - Emacs



On EOS, run the following to obtain a base configuration for Lisp:

```
cd
# Backup old files
mv .emacs .emacs.old
mv .emacs.d .emacs.d.old
# Copy base config
cp -r ~fiskse/.emacs.d-lisp .emacs.d
# Run emacs
emacs
```

# Calling Functions



- The first element of an evaluated list is the function name to be called.
- The rest of the elements are arguments to the function.
- The arguments may themselves contain lists to be evaluated.

Examples:

- `2 + 3 * 5` would be written `(+ 2 (* 3 5))`
- `a and (b or c)` would be written `(and a (or b c))`
- `foo(x, y)` (in a C-like language) would be written `(foo x y)`

# Quoting



- Quoting delays evaluation of an S-expression.
- Without quoting, the first element of a list is treated as a function.

Examples:

- `(+ 2 3) -> 5`
- `'(+ 2 3) -> (+ 2 3)`

# Defining Functions



Define functions with `defun`. The general form is:

```
(defun function-name (arguments...)
  "optional-documentation..."
  body...)
```

Optional and default parameters are also possible, but we won't go over those today.
Example:

```
(defun add (first second)
  "Add FIRST to SECOND and return the result."
  (+ first second))
```

Convention dictates that variable names be ALL CAPS in the docstring.

# Conditionals



The general form is:

```
(if cond then else...)
```

- `then` is evaluated if `cond` is not `nil`.
- `else...` is evaluated if `cond` is `nil`.
- `if` returns the value of the expression that was evaluated; either `then` or `else...`. This is a general functional programming "thing".
- To evaluate multiple expressions in `then`, use `progn` or `block`.
- `nil` is false as in other languages, and everything else is true (`t`).
- Unlike C, `0` is not considered false.

# progn



`progn` evaluates all expressions in its body and returns the result of the last one. The general form is:

```
(progn body...)
```

A contrived example of the use of `progn` with `if`:

```
(if (player-won)
  (progn
    (record-winner player)
    (1+ wins))
  wins)
```

# Defining Variables



`let`:

- Allows definition of multiple variables that are accessible within the scope of the body. The general form is:

```
(let ((var1 value) (var2 value) ... (varn value))
  body...)
```

- The last statement in the body is returned.

`defvar`:

- Defines a global variable. The general form is:

```
(defvar name init-value)
```

- These variables are global! So don't use this often.
- Does not fit the functional paradigm.

# Practice Problems: `my-square`



Define a function `my-square` that returns the square of the argument.
Here is a skeleton:

```
(defun my-square (num)
  "Return the square of NUM."
  ;; Works only for 5 and -5 :)
  25)
```

# Practice Problems: `my-square`: Solution



Possible solution:

```
(defun my-square (num)
  "Return the square of NUM."
  (* num num))
```

# Practice Problems: my-count



Define a recursive function that returns the number of elements in a list. You can't use `(length)`!.
Here is a skeleton:

```
(defun my-count (list)
  "Return the number of elements in LIST."
  ;; Works only for lists of length 10 :)
  10)
```

Expected results:

```
(my-count nil) -> 0
(my-count '()) -> 0
(my-count '(a b c)) -> 3
(my-count '(i get the point)) -> 4
```

# Practice Problems: my-count: Solution



Possible solution:

```
(defun my-count (list)
  "Return the number of elements in LIST."
  (if (car list)
      (1+ (my-count (cdr list)))
    0))
```

# Practice Problems: `my-last`



Define a recursive function `my-last` that returns the last element in a list. You can't use `(last)`! Don't forget the docstring.

Expected results:

```
(my-last nil) -> nil
(my-last '(hello world)) -> (world)
(my-last '(a b c d)) -> (d)
```

# Practice Problems: `my-last`: Solution



Possible solution:

```
(defun my-last (list)
  "Return the last cons cell of LIST.
Should behave similar to ‘last’."
  (let ((next-cell (cdr list)))
    (if next-cell
        (my-last next-cell)
      list)))
```

# Practice Problems: `my-reverse`



Define a recursive function `my-reverse` that returns a copy of the list in reverse order. You can't use `(reverse)`!
Expected results:

```
(my-reverse nil) -> nil
(my-reverse '(a)) -> (a)
(my-reverse '(a b c d)) -> (d c b a)
```

# Practice Problems: `my-reverse`: Solution



Possible solution:

```
(defun my-reverse (list)
  "Return a reversed copy of LIST."
  (if (cdr list)
      (append (last list) (my-reverse (butlast list))
    list))
```

# Do people actually use Lisp?



**Yes!**

- **AutoLISP:** The integrated scripting language for AutoCAD and other Autodesk products.
- **Mirai:** A 3D graphics suite used to animate Gollum's face in The Lord of the Rings.
- **Dynamic Analysis and Replanning Tool (DART):** An AI research project part of the United States Defense Advanced Research Projects Agency (DARPA).
- **The Emacs Editor:** 80% is written in Emacs Lisp, adding up to an insane 1,131,162 lines of Emacs Lisp code!

# Credits

Written by:


- Sean Fisk <fiskse@mail.gvsu.edu>
- Jordon Biondo <biondoj@mail.gvsu.edu>


The following free and open-source software was used to produce this presentation:
LaTeX, Beamer, Biber, Biblatex, Minted, Pygments, Emacs, AUCTeX, SCons, Clisp, cloc, Graphviz

# References



Chassell, Robert J. (Oct. 28, 2009). *An Introduction to Programming in Emacs Lisp*. 3.10. Free Software Foundation. 51 Franklin Street, Fifth Floor; Boston, MA 02110-1301 USA: GNU Press. ISBN: 1-882114-43-4.

Emacs Contributors (2012). *GNU Emacs Lisp Reference Manual*. Free Software Foundation.

Nau, Dana (Feb. 16, 2010). *Introduction to Lisp*. URL: http://www.cs.umd.edu/~nau/cmsc421/lisp-intro.pdf (visited on 03/01/2013).

Wikipedia (2013). *Lisp (programming language) — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=Lisp_(programming_language)&oldid=540157219 (visited on 03/01/2013).