Lisp!

Mark Maloof
Department of Computer Science
Georgetown University
Washington, DC 20057

September 12, 2017

Why Lisp?

- Great for symbolic computation
- Great for functional programming
- Learning Lisp will make you a better programmer in other languages

Why Lisp?

- Great for symbolic computation
- Great for functional programming
- Learning Lisp will make you a better programmer in other languages
- Phillip Greenspun's Tenth Rule of Programming: "Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified bug-ridden slow implementation of half of Common Lisp."

Symbolic Computation

- Differentiation:
  - General form: $a^d \Rightarrow da^{d-1}$
  - Application: $n^2 \Rightarrow 2n$
- DeMorgan's Law:
  - General form: $\neg(\phi \wedge \psi) \Rightarrow \neg\phi \vee \neg\psi$
  - Application: $\neg(\neg p \wedge q \rightarrow r) \Rightarrow \neg\neg p \vee \neg(q \rightarrow r)$

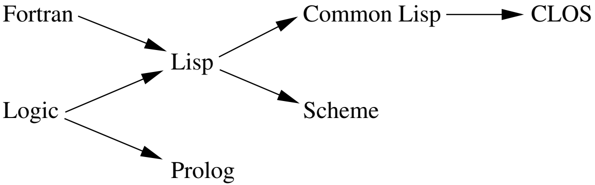
Functional Programming Languages

- Everything is a function that returns a value, even for loops
- First-class and higher-order functions
- Purely functional languages have immutable data structures, which eliminates side effects
- No side effects means computation can be distributed
- Pure functional: Haskell
- “Impure” functional: Common Lisp, Clojure
- Languages have different degrees of purity.

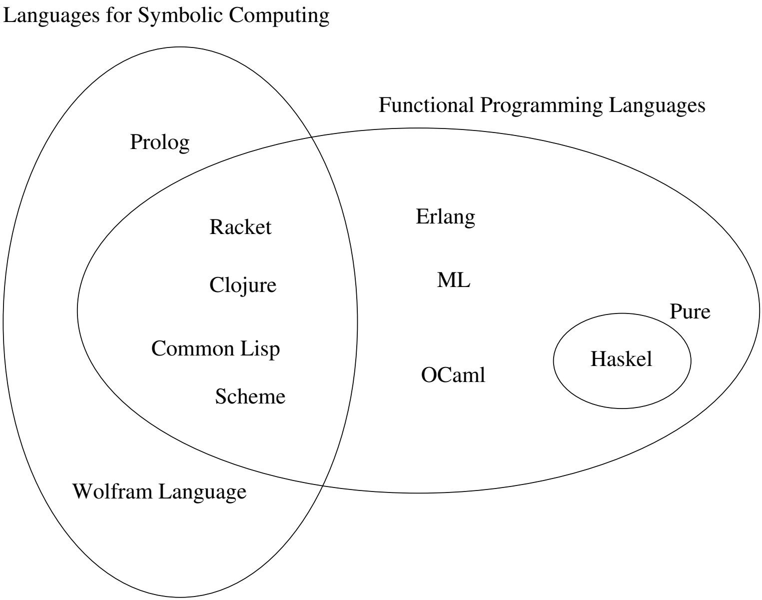
What is Lisp?

- Specified by John McCarthy in 1958
- LISP $\equiv$ LISt Processing
- Functional programming language
- Great for symbolic computing; important for AI tasks of the day
  - Predominant languages of the day were Fortran and COBOL
- Interpreted, but it can be compiled
- Great for rapid prototyping

A Lineage of Lisp



Languages for Symbolic Computing



Why Aren't We Learning Clojure?

- Clojure is a dialict of Lisp that runs on the Java Virtual Machine
- Immutable data structures
- Optimized for concurrency via map-reduce
- There are oddities (e.g., `doseq` returns `nil`)
- Clojure is still developing
- My advice: Go learn it after this class

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies
- Most everything is a function that returns a value

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies
- Most everything is a function that returns a value
  - OK, that's fine; so, no void types. I got it.

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies
- Most everything is a function that returns a value
  - OK, that's fine; so, no void types. I got it.
- Functions, function calls, and control structures are also linked lists

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies
- Most everything is a function that returns a value
  - OK, that's fine; so, no void types. I got it.
- Functions, function calls, and control structures are also linked lists
  - Umm, but, aren't linked lists for storing and manipulating data?

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies
- Most everything is a function that returns a value
  - OK, that's fine; so, no void types. I got it.
- Functions, function calls, and control structures are also linked lists
  - Umm, but, aren't linked lists for storing and manipulating data?
- Consequently, there is no distinction between code and data

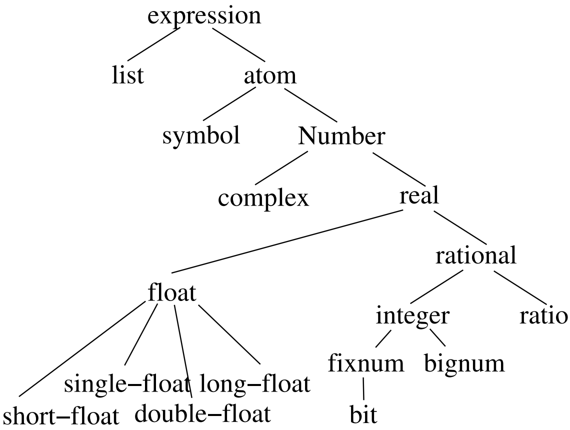
Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies
- Most everything is a function that returns a value
  - OK, that's fine; so, no void types. I got it.
- Functions, function calls, and control structures are also linked lists
  - Umm, but, aren't linked lists for storing and manipulating data?
- Consequently, there is no distinction between code and data
  - OK, now you lost me

Things to get your Head Around

- Most everything in Lisp is a list (a linked list)
  - OK, no big deal; I know about linked lists
- Lists can store elements of different types
  - OK, that's cool; I know about generic programming and class hierarchies
- Most everything is a function that returns a value
  - OK, that's fine; so, no void types. I got it.
- Functions, function calls, and control structures are also linked lists
  - Umm, but, aren't linked lists for storing and manipulating data?
- Consequently, there is no distinction between code and data
  - OK, now you lost me
- And then there are the parentheses...

Lisp Data Types

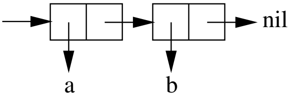


Lists and Function Calls

- We form lists and function calls using parentheses
- Prefix notation:
  - The first element of a list is the operator or function name
- Use a single quote to prevent evaluation
  - Empty list: `()`
  - List: `'(sqrt 4)`
  - Function Call: `(sqrt 4)`
  - If-statement: `(if pig 'cow 'dog)`
  - List: `'(if pig 'cow 'dog)`

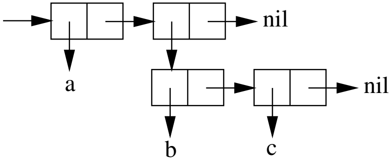
Linked Lists in Lisp

```
'(a b)
```



Linked Lists in Lisp

```
'(a (b c))
```



Numeric Operators and Functions

- `+`
- `-`
- `*`
- `sqrt`
- `1+`
- `1-`
- `min`
- `max`
- `expt`
- `log`
- `abs`
- `sin`, `cos`, `tan`
- `asin`, `acos`, `atan`
- `floor`, `ceiling`, `round`, `truncate`

Output Functions

- `(format` ⟨*file-stream*⟩ ⟨*control-string*⟩ ⟨*variable-list*⟩ `)`
- Example: `(format t "This is a test, ~a~%", 'bob)`
- Output: `This is a test, BOB`
- Basic control codes:
  - `~a` — any lisp object
  - `~s` — s-expression
  - `~d` — decimal
  - `~f` — float
  - `~%` — newline character
- Note that in sbcl ouptut is buffered by default.
- `(flush-output)`.

`prog1` Blocks

- Syntax:

  `(prog1`
  $\langle form_1 \rangle$
  $\langle form_2 \rangle$
  `...`
  $\langle form_n \rangle$`)`

- Semantics:
  - Evaluates forms 1–n
  - `prog1` evaluates to the valuation of $\langle form_1 \rangle$

`progn` Blocks

- Syntax:

  (progn
  $\langle \textit{form}_1 \rangle$
  $\langle \textit{form}_2 \rangle$
  ...
  $\langle \textit{form}_n \rangle$)

- Semantics:
  - Evaluates forms 1–n
  - `progn` evaluates to the valuation of $\langle \textit{form}_n \rangle$

Defining Functions: `defun`

- Syntax:

  (defun ⟨*function-name*⟩ (⟨*parameter-list*⟩)
      ⟨*form*⟩)

- Semantics:
  - Defines a function with the name ⟨*function-name*⟩ with the parameters ⟨*parameter-list*⟩ and the body ⟨*form*⟩.
  - Evaluates to the function ⟨*function-name*⟩

Binding Local Variables: `let` and `let*`

- Syntax:

  (`let[*]` (($\langle \textit{variable}_1 \rangle$ $\langle \textit{value}_1 \rangle$)
  ($\langle \textit{variable}_2 \rangle$ $\langle \textit{value}_2 \rangle$)
  ...
  ($\langle \textit{variable}_n \rangle$ $\langle \textit{value}_n \rangle$))
  $\langle \textit{form} \rangle$)

- Semantics:
  - Defines locally scoped variables 1–n
  - Binds or assigns values 1–n to variables 1–n, respectively
  - Evaluates $\langle \textit{form} \rangle$
  - `let` evaluates to the valuation of $\langle \textit{form} \rangle$
  - `let*` forces the sequential assignment of variables 1–n

Branching: `if-then-else`, `when`, `unless`

- Syntax:

  (if ⟨*test*⟩
  ⟨*then-form*⟩
  [⟨*else-form*⟩])

- Semantics:
  - Evaluates ⟨*test*⟩
  - If ⟨*test*⟩ is not `nil`, evaluates ⟨*then-form*⟩
  - Otherwise, if ⟨*else-form*⟩ is not `nil`, evaluates ⟨*else-form*⟩
  - `if` evaluates to the form evaluated
- Also:
  - (when ⟨*test*⟩ ⟨*form*⟩) ≡ (if ⟨*test*⟩ ⟨*then-form*⟩)
  - (unless ⟨*test*⟩ ⟨*form*⟩) ≡ (if ⟨*test*⟩ nil ⟨*else-form*⟩)
  - (unless (not ⟨*test*⟩) ⟨*form*⟩) ≡ (if (not ⟨*test*⟩) ⟨*then-form*⟩)

Relational and Equality Operators and Functions

- `equalp` — same expression?
- `equal` — same expression?
- `eql` — same symbol or number?
- `eq` — same symbol?
- = — same number?
- `atom` — is it an atom?
- `numberp` — is it a number?
- `consp` — is it a cons?
- `listp` — is it a list?
- `symbolp` — is it a symbol?
- `boundp` — is it a bound symbol?
- Also `member`, `null`, `zerop`, `plusp`, `minusp`, `evenp`, `oddp`, <, <=, >=, and >.

Nested if-then-else Statements: `cond`

- Syntax:

  (cond ($\langle test_1 \rangle$ $\langle consequent_1 \rangle$)
  ($\langle test_2 \rangle$ $\langle consequent_2 \rangle$)
  ...
  ($\langle test_n \rangle$ $\langle consequent_n \rangle$))

- Semantics:
  - Evaluates tests 1–n
  - Evaluates the consequent of the first test that is not `nil`
  - `cond` evaluates to the valuation of the consequent or to `nil` if no test is not `nil`
  - Note that the literal `t` always evaluates to true

Looping through Lists: `dolist`

- Syntax:

  `(dolist (`⟨*variable*⟩ ⟨*list-form*⟩ `[`⟨*result-form*⟩`])`
  ⟨*form*⟩`)`

- Semantics:
  - Iterates once for each element in ⟨*list-form*⟩
  - Assigns the element to ⟨*variable*⟩
  - Evaluates ⟨*form*⟩
  - Upon loop termination, evaluates ⟨*result-form*⟩
  - `dolist` evaluates to the valuation of ⟨*result-form*⟩

Count-controlled Loops: `dotimes`

- Syntax:

  (dotimes ($\langle$*variable*$\rangle$ $\langle$*upper-bound-form*$\rangle$ [$\langle$*result-form*$\rangle$])
  $\langle$*form*$\rangle$)

- Semantics:
  - Iterates over $[0, \langle \textit{upper-bound-form} \rangle)$
  - Assigns the counter value to $\langle$*variable*$\rangle$
  - Evaluates $\langle$*form*$\rangle$
  - Upon loop termination, evaluates $\langle$*result-form*$\rangle$
  - `dotimes` evaluates to the valuation of $\langle$*result-form*$\rangle$

General Looping Construct: `do` and `do*`

- Syntax:

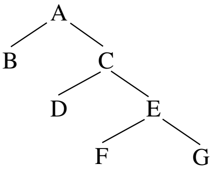
```
(do[*]((⟨var1⟩ ⟨init1⟩ ⟨update1⟩)
       (⟨var2⟩ ⟨init2⟩ ⟨update2⟩)
       ...
       (⟨varn⟩ ⟨initn⟩ ⟨updaten⟩))
      (⟨termination-test⟩ ⟨result-form⟩)
   ⟨form⟩)
```

- Semantics:
  - Defines locally scoped variables $\langle var_i \rangle$
  - Initializes them to $\langle init_i \rangle$
  - Loops until ⟨*termination-test*⟩ is not `nil` by
    - updating $\langle var_i \rangle$ by evaluating $\langle update_i \rangle$
    - evaluating ⟨*form*⟩
  - Upon loop termination, evaluates ⟨*result-form*⟩
  - `do` evaluates to the valuation of ⟨*result-form*⟩
  - `do*` forces the sequential assignment of $\langle var_i \rangle$

Arrays

- Creation
  - `(setf single (make-array '(4)))`
  - `(setf double (make-array '(2 4)))`
  - `(setf double (make-array '(2 4) :initial-element nil)) ;`
- Access
  - `(format t "~a~%" (aref single 1))`
  - `(format t "~a~%" (aref double 0 0))`
- Assignment
  - `(setf (aref single 1) 'hoya)`
  - `(setf (aref double 0 0) 'saxa)`

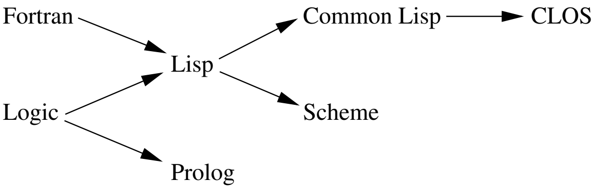
Binary Trees




```
(A (B nil nil)
   (C (D nil nil)
      (E (F nil nil)
         (G nil nil))))

(A (B)
   (C (D)
   (E (F)
      (G))))
```

Graphs




```
(setf (get 'fortran 'influenced) '(lisp))
(setf (get 'logic 'influenced) '(lisp prolog))
(setf (get 'lisp 'influenced) '(scheme common-lisp))
(setf (get 'common-lisp 'influenced) '(clos))
```

Things We Won't Cover

- Dotted Lists
- Types
- Strings
- Serious File I/O
- Packages
- Hash Tables
- Structures (aka records)
- Exceptions: throw and catch
- Programmer-defined macros
- CLOS: classes, properties, methods
- Many, many functions, forms, and macros