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: $\neg(\phi \land \psi) \Rightarrow \neg\phi \lor \neg\psi$
  - Application: $\neg(\neg p \land q \rightarrow r) \Rightarrow \neg\neg p \lor \neg(q \rightarrow r)$
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 ≡ 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

Functional Programming Languages

Wolfram Language
Prolog
Haskel
Racket
Scheme
Clojure
Common Lisp
ML
Erlang
OCaml
Pure
Haskel
Why Aren’t We Learning Clojure?

- Clojure is a dialect 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

- 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...
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

expression
  list
  atom
    symbol
    Number
      complex
      real
        rational
        integer
        ratio
          fixnum
          bignum
            bit
          short-float
          double-float
            long-float
            single-float
            float
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)

\[\text{nil}\]
'(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 output is buffered by default.
▷ (flush-output).
prog1 Blocks

- Syntax:
  
  \[
  \text{(prog1} \langle form_1 \rangle \langle form_2 \rangle \ldots \langle form_n \rangle)\]

- Semantics:
  - Evaluates forms 1–\(n\)
  - prog1 evaluates to the valuation of \(\langle form_1 \rangle\)
progn Blocks

- Syntax:
  \[
  (\text{progn}
   \langle form_1 \rangle
   \langle form_2 \rangle
   \ldots
   \langle form_n \rangle)
  \]

- Semantics:
  - Evaluates forms 1–\(n\)
  - \texttt{progn} evaluates to the valuation of \(\langle form_n \rangle\)
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[*] (⟨variable₁⟩ ⟨value₁⟩)
  (⟨variable₂⟩ ⟨value₂⟩)
  ...
  (⟨variableₙ⟩ ⟨valueₙ⟩))
⟨form⟩

▶ Semantics:

▶ Defines locally scoped variables 1–ₙ
▶ Binds or assigns values 1–ₙ to variables 1–ₙ, respectively
▶ Evaluates ⟨form⟩
▶ let evaluates to the valuation of ⟨form⟩
▶ let* forces the sequential assignment of variables 1–ₙ
Branching: if-then-else, when, unless

- Syntax:

\[
(\text{if } \langle \text{test} \rangle \\
\langle \text{then-form} \rangle \\
[\langle \text{else-form} \rangle])
\]

- Semantics:
  - Evaluates \langle test \rangle
  - If \langle test \rangle is not nil, evaluates \langle then-form \rangle
  - Otherwise, if \langle else-form \rangle is not nil, evaluates \langle else-form \rangle
  - if evaluates to the form evaluated

- Also:
  - \((\text{when } \langle \text{test} \rangle \langle \text{form} \rangle) \equiv (\text{if } \langle \text{test} \rangle \langle \text{then-form} \rangle)\)
  - \((\text{unless } \langle \text{test} \rangle \langle \text{form} \rangle) \equiv (\text{if } \langle \text{test} \rangle \text{ nil } \langle \text{else-form} \rangle)\)
  - \((\text{unless } (\text{not } \langle \text{test} \rangle) \langle \text{form} \rangle) \equiv (\text{if } (\text{not } \langle \text{test} \rangle) \langle \text{then-form} \rangle)\)
Relational and Equality Operators and Functions

- `equalp` — same expression?
- `equal` — same expression?
- `eq1` — 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: \texttt{cond}

**Syntax:**

\[
\text{cond}(\langle \text{test}_1 \rangle \langle \text{consequent}_1 \rangle) \\
\quad \text{(} \langle \text{test}_2 \rangle \langle \text{consequent}_2 \rangle \text{)} \\
\quad \ldots \\
\quad \text{(} \langle \text{test}_n \rangle \langle \text{consequent}_n \rangle \text{)}
\]

**Semantics:**

- Evaluates tests 1–\(n\)
- Evaluates the consequent of the first test that is not \texttt{nil}
- \texttt{cond} evaluates to the valuation of the consequent or to \texttt{nil} if no test is not \texttt{nil}
- Note that the literal \texttt{t} always evaluates to \texttt{true}
Looping through Lists: dolist

▶ Syntax:

\[
\text{(dolist (⟨variable⟩ ⟨list-form⟩ [⟨result-form⟩])}
\]

\[
\langle \text{form} \rangle
\]

▶ 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:**
  
  \[
  \text{(dotimes (}\langle variable\rangle \langle upper-bound-form\rangle \ [\langle result-form\rangle]) \langle form\rangle)\]

- **Semantics:**
  - Iterates over \([0, \langle upper-bound-form\rangle)\)
  - 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:

\[
\text{(do[*]}(( \langle \text{var}_1 \rangle \langle \text{init}_1 \rangle \langle \text{update}_1 \rangle )
\text{(do[\ldots]})
\text{. \ldots .})
\text{(do[*]}(( \langle \text{var}_n \rangle \langle \text{init}_n \rangle \langle \text{update}_n \rangle )
\text{(termination-test} \langle \text{result-form} \rangle )
\text{(form)})
\]

▶ Semantics:

▶ Defines locally scoped variables \( \langle \text{var}_i \rangle \)
▶ Initializes them to \( \langle \text{init}_i \rangle \)
▶ Loops until \( \langle \text{termination-test} \rangle \) is not \( \text{nil} \) by
  ▶ updating \( \langle \text{var}_i \rangle \) by evaluating \( \langle \text{update}_i \rangle \)
  ▶ evaluating \( \langle \text{form} \rangle \)
▶ Upon loop termination, evaluates \( \langle \text{result-form} \rangle \)
▶ do evaluates to the valuation of \( \langle \text{result-form} \rangle \)
▶ do* forces the sequential assignment of \( \langle \text{var}_i \rangle \)
Arrays

- **Creation**
  - `(setf single (make-array '(4)))`
  - `(setf double (make-array '(2 4)))`
  - `(setf double (make-array '(2 4) :initial-element nil)) ; sbcl`

- **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))))
(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