Introduction to Lisp
John McCarthy

- 1955: Developed the phrase “Artificial Intelligence”
- 1959: Invented “Garbage Collection”
- 1960: First Lisp Implementation
- 1961: First to publicly advocate cloud computing!
- 1971: Won the Turing Award
History

- Short for “LISt Processing”
- Timeline\(^1\) of the first high-level programming languages: Lisp is second oldest

<table>
<thead>
<tr>
<th>Year</th>
<th>Language</th>
<th>Still used today</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1957</td>
<td>FORTRAN</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>1958</td>
<td>LISP</td>
<td>Yes</td>
<td>Inspired by the (\lambda)-calculus(^2)</td>
</tr>
<tr>
<td>1958</td>
<td>ALGOL</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>1959</td>
<td>COBOL</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>1962</td>
<td>APL</td>
<td>Yes (Dyalog APL)</td>
<td></td>
</tr>
<tr>
<td>1962</td>
<td>Simula</td>
<td>No</td>
<td>First OOP</td>
</tr>
<tr>
<td>1967</td>
<td>BCPL</td>
<td>No</td>
<td>Precursor to C</td>
</tr>
</tbody>
</table>


\(^2\) A mathematical model of computation. Ref: Recursive Functions of Symbolic Expressions and Their Computation By Machine, Part I
Introduction

- Lisp has many dialects
  - Common Lisp (which we will study)
  - Scheme
  - Racket
  - Arc
  - Clojure (runs on JVM and CLR)
  - Mathematica (my favorite)
  - ...
Introduction

- In 1960’s it had:
  - Garbage Collection
  - Closures (aka lambda functions or anonymous methods)
  - Scheme had continuations (aka “yield” in C#)
  - ...

- These features were only recently introduced to “modern” programming languages

- The more a language evolves, the closer it gets to Lisp!

- In fact, when a language evolves “enough”, it might actually be considered just a dialect of Lisp!
Lisp is Unique

- It is a *programmable* programming language
  - The scanner and parser can be modified in runtime (reader macros)
  - Can implement any language on top of Lisp (similar to DSL's\(^3\))
  - Very popular for rapid prototyping in the programming language research community
    - Until now, Scheme and Racket are still very popular
  - In applications, you develop *on top of Lisp* the language suitable to your application
    - This is why the famous Common Lisp book is called “*On Lisp*”

\^3Domain Specific Languages. Think of LINQ.
Lisp

- Made only of expressions (not statements)
  - Every expression has a value
- An expression is either
  - An atom
    - A symbol: `foo` or `+`
    - A number: `3`, `5/3`, `0.745`
  - A pair (also called CONS cell): `(foo . bar)`
- That’s it!
Symbols

- Symbols are not exactly like variables
- A symbol is a name
- May have a value associated with it (bound to value)
- Or may be unbound
  - This is the key to symbolic computation
  - Think of it as the $x$ in $x^2 + 1$. $x$ has no value; it is just a symbol
CONS cells

- \((x . y)\) is a memory object holding two pointers to \(x\) and \(y\)
- \(x\) and \(y\) can be atoms or other CONS cells
CONS cells

- Can be used to represent any thing
- A binary tree: \(((a . b) . (c . d))\)
- A singly-linked list: \((a . (b . (c . ())))\)

- Lists can be expressed in a shorter way as S-expressions (but they are still singly-linked CONS cells)
  - ()
  - (foo)
  - (foo bar)
  - Can contain sublists: \((a b (c d e) d)\)
S-expressions

- Short for Symbolic Expressions
- They are a notation for nested trees
- A S-expression is either
  - An atom $x$
  - A list $(x \ y \ z)$
- To distinguish CONS cells from S-expressions, a CONS cell has a . in the middle, and exactly two parts
S-expressions as Code

- Uses Polish Notation: operation comes first, not in the middle
  - (function-name arg-1 arg-2 arg-3)
- To add 1 and 3 and 6 and 2:
  - (+ 1 3 6 2)
S-expressions as Code

- To compute \( (1+2*3*4)/5 \)
  - In Lisp: \( (/ (+ 1 (* 2 3 4)) 5) \)
  - Is equivalent to the abstract syntax tree of \( (1+2*3*4)/5 \)
S-expressions as Data

- Any S-expression is evaluated immediately by Lisp
  - (+ 1 2) will be transformed to 3 instantly before it is passed to other pieces of code
- Lisp will try to evaluate a data list (1 2 3) but will fail, since 1 is not a function
- There must be way to stop Lisp from trying to evaluate (1 2 3)
  - (quote (1 2 3)) returns (1 2 3) unevaluated
- Can be equivalently written as ’ (1 2 3)
$M$-expressions

- $M$-expressions (meta-expressions)
  - $S$-expressions were intended as a low-level representation
  - The higher-level representation, $M$-expressions were never implemented
  - Some Lisp dialects, such as Mathematica, implement $M$-expressions
Lisp Primitives
Lisp Primitives

- Lisp can be defined using as few as 8 primitives (think of them as axioms)
- Every Lisp operation can be defined using only these primitives
- In the 70’s, MIT developed a Lisp Machine
- Its machine code supported the Lisp primitives on hardware level
Primitive #1: **quote**

- `(quote x)` returns `x` as it is *without evaluation*
- Can also be written as `'x`

```
(quote a)
=> a

'a
=> a

(quote (+ 1 2))
=> (+ 1 2)
```
Primitive #2: atom

- `(atom x)` will evaluate `x`
- Then it will returns true if the value of `x` is an atom or the empty list
- Otherwise it will return false
- True is denoted by the atom `t`
- False is denoted by the empty cell `()` (or equivalently, `nil`)
Primitive #1: *quote, revisited*

- Quote is to prevent evaluation
- To represent code as data

```
(quote (+ 1 2))
=> (+ 1 2)

(atom (atom 'a))
=> t

(atom '(atom 'a))
=> nil
```
Primitive #3: `eq`

- `(eq x y)` evaluates `x` and `y` and returns `t` only if
  - The value of `x` and the value of `y` are the same `atom`
  - The value of `x` and the value of `y` are both the empty list `()`

```lisp
(eq 'a 'a)
=> t

(eq 'a 'b)
=> nil

(eq '() '())
=> t

(eq (+ 1 3) (+ 2 2))
=> t
```
Primitives #4 and #5: car and cdr – reading CONS cells

- A CONS cell (x . y) is composed of two parts
  - (car '(x . y)) evaluates to x
  - (cdr '(x . y)) evaluates to y
- Recall: (a b c) is represented as
  (a . (b . (c . nil)))

```
(car '(a b c))
=> a
(cdr '(a b c))
=> (b c)
```
Primitives #4 and #5: \texttt{car} and \texttt{cdr}

- \texttt{car} can be thought of as \texttt{first} of a list
- \texttt{cdr} can be thought of as \texttt{rest} of a list
- Name comes from IBM 704 computer
  - \texttt{car}: Contents of the \texttt{Address Register} number
  - \texttt{cdr}: Contents of the \texttt{Decrement Register} number
- It was proposed to rename them to \texttt{first} and \texttt{rest}
- But they remained because the name is composable
  - Let \( v \) be equal to \( '((a\ b)\ (c\ d)) \)
  - \((\texttt{caar}\ v) = (\texttt{car}\ (\texttt{car}\ v)) = a\)
  - \((\texttt{cadr}\ v) = (\texttt{car}\ (\texttt{cdr}\ v)) = (c\ d) \) (think of it as \texttt{second})
  - \((\texttt{cadar}\ v) = b\)
  - \((\texttt{cdar}\ v) = (b)\)
  - ...
Primitive #6: \textit{cons}

\begin{itemize}
  \item \texttt{(cons x y) \textit{cons}tructs a CONS cell}
\end{itemize}

\begin{verbatim}
(cons 'x 'y)
=> (x . y)
(cons 'x '(y z))
=> (x y z)
\end{verbatim}
Primitive #7: \texttt{cond}: Conditional evaluation

\begin{itemize}
  \item \texttt{(cond (p1 e1) (p2 e2) (p3 e3))}
  \item If \( p_1 \) is true, evaluates to \( e_1 \),
  \item Else if \( p_2 \) is true, evaluates to \( e_2 \),
  \item Usually the last condition is \texttt{t}
\end{itemize}

\begin{verbatim}
(cond ((< x 0) 'negative)
     ((eq x 0) 'zero)
     (t 'positive))
\end{verbatim}
Primitive #8: Defining Functions: `lambda`

- `(lambda (x) (car x))` is a function that takes one parameter: `x` and returns the first element of `x`
  - What happens if `x` is an atom?
    - Runtime error. Lisp is dynamically typed

- `(((lambda (x) (car x)) '(a b))` is how the function is called on the argument `(a b)`
Assigning Values to Symbols

- Indirectly using lambda ($\beta$-reduction in $\lambda$-calculus)

(((lambda (x y z)
  (+ x y z))
1 2 3)

- This is one way let could be implemented (using Macros!)

(let ((x 1)
  (y 2)
  (z 3))
 (+ x y z))
Common Lisp
The Two Namespaces in Common Lisp

- In Common Lisp a symbol has *two values*
  - Variable value: + has the value 3 (for example)
  - Function value: + has the function value: (the address of the function that does addition)
- Can access both of them like this: (symbol-value ’+) and (symbol-function ’+)
- So when you name variables you do not worry they would have the same name as a function
- Function namespace is used only in two cases
  - When evaluating the first element in a list: (f a b c)
  - When using (function f), or #’f for short
- Variable namespace is used everywhere else

```
(let ((+ 1)
      (* 2))
  (+ + *))
=> 3
```
Functions: `defun`

```
(defun f (arg1 arg2)
  body)

(defun factorial (n)
  (if (zerop n)
      1
      (* n (factorial (- n)))))
```
Functions: how is `defun` implemented

```lisp
(defun mult-2 (x)
  (* 2 x))
```

is the same thing as

```lisp
(setq (symbol-function 'mult-2)
      (lambda (x) (* 2 x)))
```
Passing Functions as Arguments

```lisp
(defun some-function (f)
  (funcall f '(b c)))

(defun prepend-an-a (x)
  (cons 'a x))

(some-function #'prepend-an-a)
=> (a b c)
```

- The function #'prepend-an-a is passed into some-function as a variable value, not a function value
  - That is, (symbol-value f) is what is bound, not (function-value f)
- This is why funcall is needed
- funcall uses the variable value of the symbol f
Now let’s write some functions!
List length

```lisp
(defun list-size (list)
  (cond ((eq list nil) 0)
        (t (1+ (list-size (cdr list)))))

(list-size '()) => 0
(list-size '(1)) => 1
(list-size '(1 2)) => 2
(list-size '(1 2 3)) => 3
(list-size '(1 (2 3) 4)) => ?
```
Append

(defun append-to-list (list item)
  (cond ((eq list nil) (cons item nil))
        (t (cons (car list) (append-to-list (cdr list) item))))
)

(append-to-list '(a) 'b) => (a b)
(append-to-list '() '()) => (nil)
(append-to-list '(1 2) 3) => (1 2 3)
(append-to-list '(1) 2) => (1 2)
(append-to-list '() 2) => (2)
(append-to-list '(1) '(2 3)) => ?
iota

(defun iota (n) (loop for i from 1 to n collect i))

We may see later how to implement loop this way using macros
mapcar

- To apply a function to every element in a list

(mapcar #'1+ '(1 2 3)) => (2 3 4)
Exercises

- Join two lists
- Set membership
- Reverse a list
Quick Sort

(defun qs (list)
  (if (cdr list) ;; length > 1?
      (flet ((pivot (test)
               (remove (car list) list :test-not test)))
        (append (qs (pivot #’>)) (pivot #’=) (qs (pivot #’<)))
        list))
Symbolic Differentiation

- `expt` is the raise-to-power operator in Common Lisp: \((\text{expt } 3 \ 2) = 3^2\)

- In differentiation, \(\frac{dy}{dx} \ ax^b = abx^{b-1}\)

- Can we make a program that takes Mathematical expression and differentiate them?

```lisp
(defun ^ (a b) (expt a b))
(defun diff (expr)
  (cond ((eq (car expr) '^) ;; x^b => b * x ^ (b -1)
         (let ((x (cadr expr))
               (b (caddr expr)))
          (list '* b (list '^ x (list '-' b 1)))))))

(diff '('^ x b)) => (* b (^ x (- b 1)))
(eval (subst 3 'b (subst 2 'x (diff '('^ x b))))) => 12
```
Symbolic Differentiation - Using destructuring-bind (1)

(defun diff (expr)
  (cond ((eq (car expr) '^)
             (destructuring-bind (op x b) expr
               (list '* b (list '^ x (list '- b 1)))))
```
Symbolic Differentiation - Using destructuring-bind (2)

(defun diff (expr)
  (destructuring-bind (op 1st 2nd . rest) expr
    (if (eq op '^)
      (list '*' 2nd (list '^ 1st (list '-' 2nd 1))))))
Symbolic Differentiation - dispatching

```lisp
(defun |diff-^| (x b)
  (list '* b (list '^ x (list '-' b 1))))

(defun diff (expr)
  (destructuring-bind (op . rest) expr
    (apply (intern
      (apply (concatenate 'string "diff-" (string op)))
      rest)))

(diff '(^ x b))
```
Next Lecture

- Lisp Macros
- Assignment: implement a more general symbolic differentiator
  - It should be able to differentiate any polynomial (with integer, rational, or float coefficients)
  - It should return the result in a simplified form (algebraic simplification)
    - Example: \((+ 1 (+ 1 x))\) should be simplified to \((+ 2 x)\)
Thank you!