Old-School FP
Common Lisp

Alan Dipert, alan@dipert.org

Orange Combinator

2018-12-17

Lisp History

- Invented by John McCarthy at MIT in 1958
- Inspired by Alonzo Church’s lambda calculus
- Described in 1960 paper, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*
  - Shows a Turing-complete language for algorithms and defines it in terms of an **eval** function written in itself
- Implemented by Steve Russell around 1960
  - A student of McCarthy’s, Russell saw McCarthy’s paper on his desk, read it, and implemented the **eval** function in machine code. The result was a Lisp interpreter.
- Language represented with its own data structures: **homoiconicity**

Lambda Calculus

- (λx.M)
  - **Lambda abstraction**
  - M is any **lambda term**
- β-reduction
  - ((λx.M) E) → (M[x:=E])

Lisp

- (lambda (x) M…)
  - Anonymous function
  - M… is zero or more **expressions**
- Evaluation
  - ((lambda (x) M…) E) → (M[x:=E])

Relationship to AI

- Associated with Artificial Intelligence (AI) research since its invention at the MIT AI Lab by McCarthy in 1958
- First **symbolic** programming language
  - As opposed to **numeric** like Fortran (1957)
  - Objects and operations upon them not necessarily numeric
    - Logical
    - Set-theoretic
- First **garbage-collected** language
  - Memory reclaimed automatically when possible
- Really they just needed a decent **scripting language** to iterate quickly with
- Today, most programming is “symbolic”!

Classic Lisp: member*

```
(defun member* (x xs)
  (if xs
      (if (eql (car xs) x)
          xs
          (member* x (cdr xs)))))

(member* 2 ‘(1 2 3)) → ‘(2 3)
```

- **defun**: defines a function in the current package
- **(x xs)**: function parameters
- **if**: conditional expression, (if *test consequent <alternative>*)
- **car**: accesses the first element of a **cons cell**, which is a pair
- **eql**: “shallow” equality predicate between symbols and numbers
- **cdr**: accesses the second element of a **cons cell**

Classic Lisp: member*

```
(defun member* (x xs)
  (if xs
      (if (eql (car xs) x)
          xs
          (member* x (cdr xs)))))
```

- **xs** is a list; lists are **linked lists** made up of **cons cells**, the final cell of which has a **nil cdr**
- The empty list **()** can also be written as **nil**
- **nil** is the only logically-false value
- definition is **tail-recursive** as its call to itself is the “last” expression in the function body, or in **tail position**

Classic Lisp: mapcar*

```
(defun mapcar* (f xs)
  (if xs
      (cons (funcall f (car xs))
            (mapcar* f (cdr xs)))))

(mapcar* (lambda (n) (* n n)) ‘(1 2 3)) → ‘(1 4 9)
```

- mapcar* is a higher-order function (HOF): it takes another function as an argument
- **cons**: construct a new cons cell with supplied **car** and **cdr**
- **funcall**: invoke user-supplied function **f** with the argument **(car xs)**
  - Can’t just (f (car xs)) because functions and values inhabit separate **namespaces**
  - Common Lisp is a **Lisp-2** because of this bifurcation. Scheme and Clojure are examples of **Lisp-1s**: in those Lisps, functions are values.

Classic Lisp: mapcar*

```
(defun mapcar* (f xs)
  (if xs
      (cons (funcall f (car xs))
            (mapcar* f (cdr xs)))))

(mapcar* (lambda (n) (* n n)) ‘(1 2 3)) → ‘(1 4 9)
```

- mapcar* is a **higher-order function**: it accepts a function as an argument
- mapcar* **is *not* tail-recursive**
  - Can’t be converted automatically by the compiler into a loop
- Sufficiently-long input lists will cause stack overflow
- Problem can be “solved” efficiently by building up a list of results and maintaining the last result, so that it can be appended to
- Imperative/tedious nature of such code motivates **lazy evaluation** or **lazy sequences**

Common Lisp: iterative mapcar*

```
(defun mapcar* (f xs)
  (loop for x in xs
        collect (funcall f x)))
```

- **loop**: A **macro** that implements an embedded DSL specifically tailored to iteration
- Macros are functions that receive their arguments **un-evaluated**, as **datums**, at **compile time**.
- As the compiler walks source code, it substitutes any macro it encounters with that macro's return value or **expansion**
  - Compiler repeatedly calls the macro until its expansion no longer changes
  - Macros may return code that contains calls to other macros
  - Once the expansion no longer changes, all macros have been substituted.
- Macros can be used to implement any language that can be described with Lisp data. We're **programmers**, we should **program** our **programs**.
- Rare in daily use but uniquely powerful and key to performance

Implementations

- Commercial
  - Allegro CL: integrates with persistent object store, has high-performance embedded Prolog
  - LispWorks: Looks nice for hobby-programming and delivering small executables
- Free
  - Steel Bank Common Lisp (SBCL): the premier free implementation
  - Clasp: New LLVM-based implementation in heavy development with much promise
  - JSCL: Browser-based implementation that compiles to JS
  - SLip: Browser-based implementation with its own VM, green threads http://lisperator.net/slip/

Minor Encumbrances

- Package system
- Effectively case-insensitive
  - Complicates interoperation with other languages, data formats
  - Allegro CL mitigates this with its “Modern Mode”
- Lack of cURL binding and JSON handling out of the box
  - Any work program will necessarily require QuickLisp
  - Compare to Node.js, Ruby or Python, all have built-in JSON and HTTP libraries
- Size of delivered executables
  - Only commercial implementations have the ability to **tree-shake** the **Lisp image** in order to deliver small (< 50mb) standalone executables.

Semantic Difficulties

- Plethora of equality functions
  - A double-edged sword: careful selection of equality tests is critical for performance
- “Hash tables are not Lispy”
  - Functional “build up a result” style of programming hampered by the lack of associative, immutable structures
  - fset and several other libraries offer such types, but they do not integrate directly with the language, because CL’s sequence abstraction is not extensible and its APIs were not designed with immutability in mind.
  - fset’s types are not best-in-class (HAMT as empowered by x86 **popcnt** instruction)
  - In SBCL, the sequence type is extensible.

The Awesome

- Language is standardized: it has a freaking manual you can keep at your desk
- Relationship between Lisp code and machine code is direct
  - (disassemble ‘(lambda (x) x))
- Carefully-written programs can approach performance of C
  - Type hints and other metadata about the program can be communicated to the compiler with **declaim** and **declare**
  - Numbers can be represented very efficiently as **tagged pointers**
  - **tagbody** implements GOTO
- Linked lists aren’t the coolest data structure, but *damn* can you get a lot of mileage out of them!
- First-class structures and types with **defstruct** and **defclass**
- IRC communities #lisp and #sbcl on Freenode are exceedingly helpful
- **It just has a great old-timey feel and is loads of fun to use and experiment with!**