

Pride and prejudice: Four decades of Lisp

Stuart Watt

Human Cognition Research Laboratory,
The Open University,
Walton Hall,
Milton Keynes, MK7 6AA.

Email: S.N.K.Watt@open.ac.uk

Presented to:
Workshop on the choice of programming languages
29-30th September 1993

From an MIT job advert

...Applicants must also have extensive knowledge of C and UNIX, although they should also have sufficiently good programming taste to not consider this an achievement...

Why should I teach Lisp?

Lisp is the basis for the teaching of programming at MIT. Why?

- Lisp doesn't force any single style. The same language can be used to teach procedural, data-flow, and object-oriented programming.
- Students don't have to learn a different syntax for each style of programming.
- Students learn how the different styles work, by implementing them in Lisp as well as programming them.
- Lisp has a closeness to formal language theory that helps understanding of compilers, interpreters, and language semantics.

Lisp's success as a teaching language isn't a coincidence. It has a unique simplicity and mathematical elegance that corresponds to the concepts that are taught in programming.

First-course programming language frequencies

140 Pascal

57 Ada

48 Scheme

45 Modula (52 for all dialects)

25 C

8 Fortran

7 C++

6 Modula-2

5 SML

4 Turing

... ..

Lisp: A language surrounded by myths

Many things are said of Lisp that aren't true.

- It is big.
- It is slow.
- It is difficult to learn.
- It is an “artificial intelligence” language.
- It is an academic/research language.

Twenty years ago, these were true, but Lisp has evolved, and the requirements on programming languages have changed.

- Software costs outweigh hardware costs; Lisp's fast development can reduce costs significantly.
- Lisp has changed to meet new requirements; today it has as much Pascal in its ancestry as it does traditional Lisp.

What is Lisp?

- “Lisp” is an acronym for “list processing”
- Derived from Church’s lambda calculus, but made efficiently computable
- Originally designed for symbolic processing, eg. differentiation and integration of algebraic expressions
- Data as symbolic expressions; simple, general data structures, for both programs and data.
- Consistent; everything in Lisp is a first-class object, numbers, lists, functions, symbols, objects, etc.
- Lisp has a thirty-odd year long history of success; it has a culture, and has been proved many times over.

Lisp dialects today

There are two dialects in common use today, reflecting different requirements

- Common Lisp;
 - designed as an “industrial strength” language.
 - powerful, expressive, and efficient.
- Scheme;
 - designed as a teaching language, and in common use today.
 - a small, clean “gem-like” language with Pascal in its ancestry.

These languages have many features in common.

- Both run on almost every kind of computer.
- Many public domain implementations exist.
- Both are designed to be compiled efficiently.
- Both use static binding, rather than the dynamic binding common in interpreted implementations in the past.

Data typing in Lisp

Lisp is a “dynamically typed” language; it associates the type with the value rather than the variable.

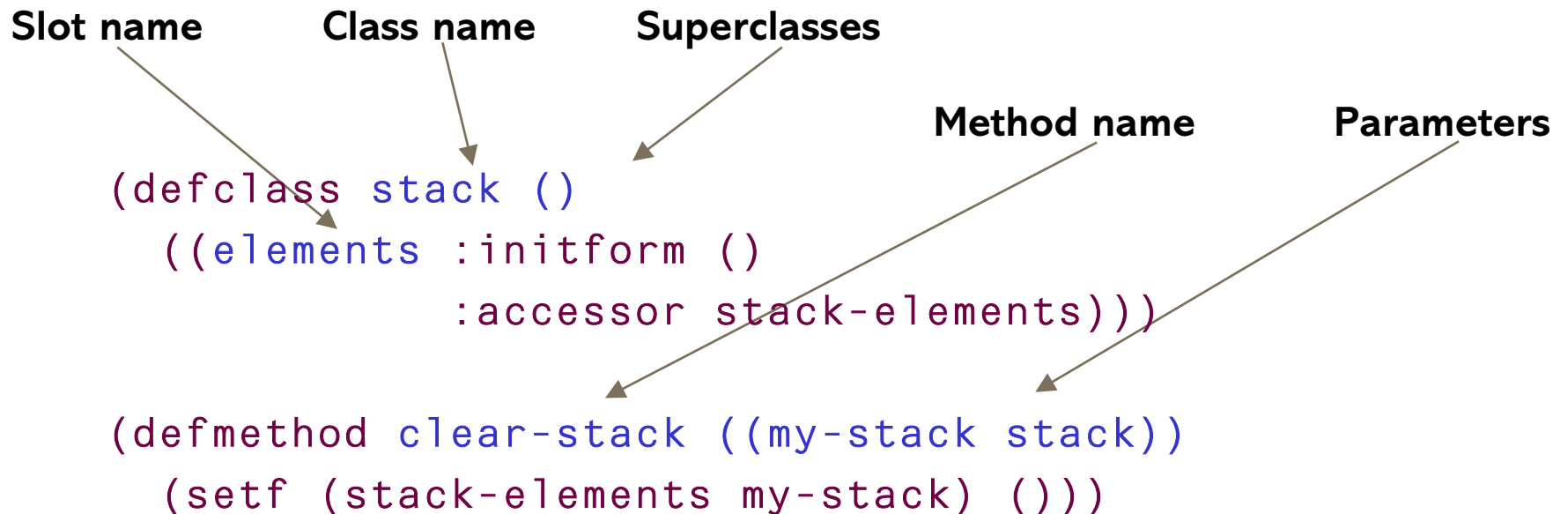
- First class data types include:
 - Symbols; `nil`, `factorial`, `+`
 - Cons cells, lists; `(1 2 3 4 5)`, `(foo . bar)`
 - Numbers; `1`, `1.63552716`, `2/3`, `66159327278283638`
 - Functions; `#'factorial`, `#'+`
 - Closures; `#<lexical-closure #x62AF40>`
 - Objects; `#<stack #x62AE20>`
 - Arrays; `"Hello there"`, `#(1 2 3 4 5)`
 - Hash tables; `#<hash-table #x61AE90>`

Detecting errors at run-time is easy; the debugger will appear whenever there is a type error.

Is Lisp object-oriented?

The Common Lisp object system is perhaps the most advanced object system in the world. It features:

- Multiple inheritance
- Multimethods; dispatching on more than one parameter



An example using CLOS

It is easy to define a stack class in CLOS:

```
(defclass stack ()
  ((elements :initform ()
             :accessor stack-elements)))

(defmethod push-element ((my-stack stack) element)
  (push element (stack-elements my-stack)))

(defmethod pop-element ((my-stack stack))
  (check-type (stack-elements my-stack) cons)
  (pop (stack-elements my-stack)))
```

<code>defclass</code>	defines a class
<code>defmethod</code>	defines a method
<code>setf</code>	assignment
<code>push/pop</code>	Common Lisp stack primitives

Beyond message-passing: multimethods

In CLOS, methods can dispatch on more than one parameter; message-passing is replaced by “generic functions”.

```
(defmethod append-list ((list1 list) (list2 list))  
  (append list1 list2))
```

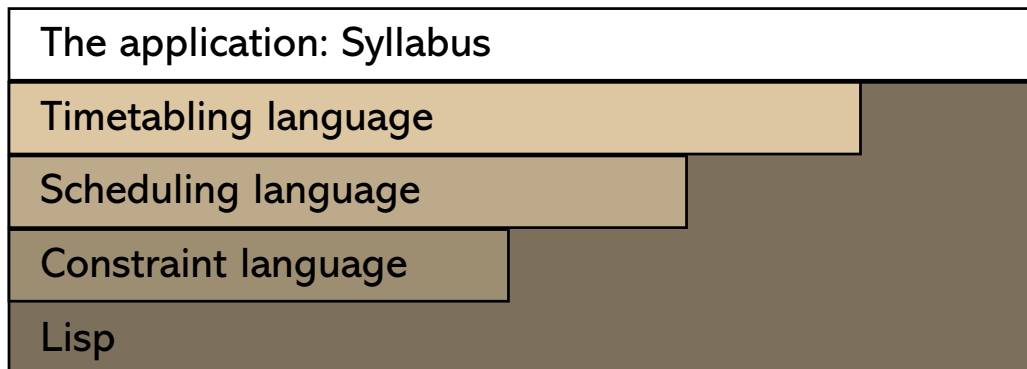
```
(defmethod append-list (list1 (list2 stack))  
  (append-list list1 (stack-elements list2)))
```

```
(defmethod append-list ((list1 stack) list2)  
  (append-list (stack-elements list1) list2))
```

Calling `(append-list stack1 stack2)` calls the third method, which calls the second method, which calls the first method, and then returns the final result.

Cascades of languages

Lisp programs often end up as cascades of languages, starting with general-purpose ones which gradually become more specialised to the task.



This has several advantages, including, for example:

- Reuse
- Reliability; ease of debugging

Languages and Lisp

Lisp makes the treatment of different languages simple; this is the core of a Lisp interpreter for Scheme.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                  (list-of-values (operands exp) env)))
        (else (error "Unknown expression" exp))))
```

Exploration of the underlying mechanism makes programs easier to understand. Lisp is a natural for this.

New dimensions of Lisp: macros and macro packages

Macros are Lisp programs that transform other Lisp programs, by operating on them as data structures.

- General purpose macro packages can be developed.
 - Series package; data-flow programming.
 - Screamer package: nondeterministic programming.

- Normal definition for the factorial function

```
(defun fact (n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

- Series package definition for the factorial function

```
(defun fact (n)
  (collect-product (scan-range :start 1 :upto n)))
```

Debugging in Lisp

Lisp has an interactive debugger. When an error happens, a window like this appears, and the inspector can be used to help find the problem.

The screenshot displays the Lisp debugger interface, divided into three main sections:

- Backtrace for Debugger Level 1:** A list of function calls in the backtrace. The current frame, `pop-element`, is highlighted in green. The list includes:
 - `procyon::signal-error-and-get-new...`
 - `*<function 5 #x15D72A>`
 - `pop-element`**
 - `*<function 0 #x17BFE0>`
 - `eval`
 - `trap-exits frame`
 - `toploop::top-eval-no-prompt`
 - `toploop::top-read-eval-print`
 - `trap-exits frame`
 - `toploop::toploop1`
 - `*<function 0 #xAB3AC>`
 - `toploop::toploop`
- Frame 6:** A central area with control buttons:
 - `More...`
 - `Open`** (highlighted)
 - `Return`
 - `Continue`
 - `Abort`
- Stack Frame 6:** A detailed view of the current stack frame:
 - Label: `pop-element executing w`
 - Content: `#<stack #x15EDC8>`
 - Separator: `#<stack #x15E1`
 - Content: `#<stack #x15EDC8> is a`
 - Content: `Class • #<standard-clas`
 - Content: `elements • nil`

Expanding the business: using Lisp in industry

- Many large companies have small groups using Lisp to solve real problems, but the prejudices still live.
- A principal advantage of Lisp is its development speed; other languages need more effort for the same result.
- Even with an incremental C++ environment, Lisp programmers of the same skill are 2–5 times more productive; this makes some projects viable which otherwise wouldn't be.
- Skill shortage is a problem, but training is straightforward and people quickly become productive.

Conclusions: the “take home” message

- Lisp is the victim of pride and prejudice.
- Lisp is proven; it has adapted and survived through its thirty-year history.
- Lisp can be—and is—used effectively as a teaching resource.
- Lisp encourages a style of cascading languages; ideal for software reuse.
- Lisp can make—and has made—some “niche” applications viable in industry.
- Lisp is still adapting to new circumstances and breaking new ground. It’ll be here for many years yet.