*LISP Ex Nihilo*

*Introduction*

***LISP Ex Nihilo*** describes the construction of a programming system from minimal primitives.

The venerable LISP programming language is perhaps over-examined in the literature, as much for its contributions to modern programming theory and practice as its age. At the risk of repeating what others have said and far better, it’s enlightening to see how a LISP can be iteratively constructed from a few basic primitives to a useful programming environment.

Much of this material was discovered when building the *Logica* programming system. I make no claims to any fundamental discoveries, but the excellent work of people like Christopher Quinniec, Paul Graham, Hal Abelson, and Gerald Sussman sometimes assume more intimate knowledge of the primitive capabilities of the systems they explore than the reader might have at their disposal.

Much of LISP is written in LISP itself, but there remains a hard primitive layer that abstracts the machine into something a LISP can be built upon. Such layers are present in all programming systems, compilers in general are built on top of other languages (most usefully, themselves.) There was a time when people like Don Knuth wrote ALGOL-58 compilers from assembly code (though he wrote flavored assemblers for the task), but today we have high performance programming language implementations for nearly every language ever in use.

Constructing a useful primitive layer is then a matter of picking an implementation language and writing the set of primitives that underly the desired language. These primitives form the run-time environment of a language, and are a means to implement the language system on a physical machine/operating system.

This work examines that primitive layer, and shows the rise of a LISP from it.

The author thanks Guy L. Steele, Jr, Peter Norvig, Henry Baker, and Doug Hoyte among many others for their contributions to the subject area, it is only by standing on the shoulders of giants like these that we advance. Parts of this work were created during my tenure as Visiting Professor of Computer Science, St. Andrews University, Laurinburg, NC from 2011 to 2016.

*James Putnam, Fall 2020*

*The lambda calculus*

***The lambda calculus gives us a way of reasoning*** about *functions*, which are at the heart of a programming language. This calculus introduces a notation and a set of fundamental operations like applications and reductions that can be employed to produce data type inferencing systems and the basic primitives of function construction and application to arguments.

While an in-depth discussion of the lambda calculus (and there are several) and combinatorial logic is outside the scope of this exposition, we’ll refer to several principles frequently.

*β-reduction*

*Special operators are primitive*

***LISP function calls are a list*** in which the first element is a *function designator* and the remaining elements, if any, are the arguments to the function call. Such lists, along with symbols, scalars, and vectors, are called *s-expressions*.

We want our LISP to handle lists as data, in specific as arguments to functions, so we need a syntax to distinguish the list (+ 2 3) from the function call (+ 2 3). Both are lists of three elements but we indicate a non-function call list by applying the *quote* operator.

(*quote* (+ 2 3))

This syntax produces a list, where a naked (+ 2 3) is interpreted as a function call.

The standard LISP function call process evaluates all arguments before applying the function. If *quote* were an actual function then the (+ 2 3) argument would be interpreted as a function call, rendering it ineffective.

*quote* is instead a *special operator*, which has special rules for application, i.e. its argument is not evaluated, and evaluating a *quote* form simply returns its argument. Quoting is so common that a syntactic shortcut for *quote* forms emerged quite early.

‘(+ 2 3) is equivalent to (*quote* (+ 2 3))

This discipline for list notation is where LISPs gain much of their bad reputation, in part because complex applications may lead to deeply nested parenthetical expressions. While a competent text editor helps unravel the nesting, other LISP-like languages like Clojure use square brackets for constructing some kinds of lists.

Since s*pecial operators* have special evaluation rules, they form the basis of LISP primitives.

*The core implementation language*

***It is tempting to implement the desired language*** in the primitive layer. While the resulting monolithic code is self-contained (for whatever virtues that has), it has a strong potential for what is referred to in database circles as an *impedance mismatch,* which are informally the kinds of problems that occur when an implementation language does not easily support the kinds of structures the desired language eventually needs.

This results in a lot of primitive code that may be more easily and succinctly expressed in a language that is closer to the end language, and offers more opportunities for code reuse.

Language implementers often construct a tower of languages, starting with a thin subset of the

desired language.

*LISP Ex Nihilo*’s core reference implementation is about 8K LOC of C++14. It implements the *lambda*, *quote*, *defvar,* and *defmacro* primitives, a garbage-collected heap, minimal reader and printer. It supports LISP-1 namespaced and lexical *symbols, functions, fixnums, floats, conses, streams, structs, exceptions,* and specialized *vectors* and is packaged in a library.

*Source transformation by Macros*

*Implications of Immutability, Mutable Lexical Symbols*

*Arithmetic Primitives*

*Special Operators*

:defcon

:lambda, rest args

:letq

;macro

:quote

*let/let\* is lambda*

*list/list\* is rest*

*Closures*

*Conditional Primitives*

*Shame :t and :nil are symbols, we could do the Smalltalk trick*

*Symbol/Keyword/Namespace Primitives*

*List/Folds Primitives*

*Vector and Specialized Vector Primitives*

*Stream Primitives*

*Struct Primitives*

*Exception Primitives*

*Read/Print Primitives*

*trampolines, eval/apply, identity*

*Environment Primitives*

*Heap/GC Primitives*

*Type Primitives, true/false*

*Macro Primitives*