The Design and Implementation of a Modern Lisp Dialect

Nicholas Alexander
Samuel Davis

May 22, 2006

Abstract

Lisp, invented in 1958 by John McCarthy, revolutionized how programs could be written and expressed. Lisp introduced many modern language features such as recursion, garbage collection, and structured conditionals. Lisp also introduced a new, shared notation for code and data, which allowed Lisp code to be manipulated as data. While mainstream languages have adapted many of Lisp's other features, the ability to express code as data remains unique to Lisp. However, the influence of the C programming model has changed the idea of what a programming language requires. The two most popular Lisp dialects, Common Lisp and Scheme, have a number of attributes that are a result of when they were conceived. Most notably, both dialects have a policy of OS-neutrality, which means that they cannot communicate with the OS at an intimate level. The result is that libraries for operations such as remote communication, file manipulation, and high-end graphics are either limited in their capabilities, platform-dependent, or unreliable. Sigma is designed to be a modern Lisp dialect that fulfills the needs of advanced programmers by allowing close interaction with the OS and providing the immense abstraction power of Lisp.

1 Introduction

1.1 Purpose

The purpose of this project is to create a new dialect of Lisp, called Sigma, which corrects the deficiencies of Common Lisp and Scheme such as forced OS neutrality, as well as its interpreter. The language will be targeted to intelligent programmers, and will provide a high level of abstraction with virtually no restrictions on the programmer.

1.2 Scope of Study

The final result is expected to be an interpreter that can accept either explicit terminal input or file input and execute it. No attempt at compilation will be made.

2 Background and Review of Literature

2.1 Background of Lisp

2.1.1 Early History

John McCarthy invented Lisp in 1958 in an attempt to make a more elegant model of computation than the Turing Machine for use in his research into artificial intelligence. It was originally a language for expressing programs as math that extended lambda calculus. Lisp first appeared as a programming language when one of McCarthy's students, Steve Russell, wrote an interpreter. Lisp was introduced in a famous paper by McCarthy entitled "Recursive Functions of Symbolic Expressions and their Computation by Machine (Part 1)". In this paper, McCarthy showed how, using a notation for code as a tree of symbols and seven primitive functions, one could implement a complete programming language.

2.1.2 Lisp at MIT

A dialect of Lisp called MacLisp was developed at MIT which was the first to incorporate macros, which were a major advancement over simple text substitutions in that they took advantage of Lisp's code-as-data notation to allow any possible manipulation of code. This development gave Lisp a permanent advantage over other languages, and is what defines Lisp today.

2.1.3 Modern Lisp

Today, most Lisp programming in done in Common Lisp, which was initially defined in Guy Steele's book Common Lisp: the Language in 1984 and standardized by ANSI in 1994 in an attempt to create a single, dominant Lisp. Another popular dialect is Scheme, invented in the 1970's, which has also enjoyed use for teaching and academic study due to its small, clean core. Both dialects, however have flaws. Common Lisp is criticized for being overly large and more difficult to learn. Scheme, by contrast, has such a small definition that it can be difficult to actually work in, and its hygienic macro system, designed to avoid unwanted variable capture, make it extremely difficult to perform intentional variable capture. When both dialects were conceived, languages were supposed to be OS-neutral, so many tasks that require talking to the OS can only be done using obscure, implementation-specific techniques.

2.2 Literature

2.2.1 McCarthy's Paper

Lisp was introduced in McCarthy's 1960 paper "Recursive Functions of Symbolic Expressions and their Computation by Machine (Part 1)." In this paper, McCarthy defined Lisp by extending lambda calculus by adding conditionals, a notation for recursive functions, and a notation for expressing arbitrary code as a list, which he called an S-expression. This paper

also marks the first appearance of the `eval` function, which was a complete interpreter for Lisp that fit on half of a page.

2.2.2 The Works of Paul Graham

Lisp has had a resurgence of interest in recent years, which can be partly attributed to the writings of Paul Graham, who made Viaweb, the first Application Service Provider, almost entirely in Common Lisp. He eventually sold Viaweb to Yahoo! for $50 million, where it became Yahoo! Store. Graham has written two books on Lisp, "ANSI Common Lisp" and "On Lisp", and also hosts numerous essays on his personal web site. Graham's writings generally do not refer specifically to Common Lisp, but rather to what he called "Lisp-the-platonic-form". Graham maintains that a new dialect of Lisp, with the ability to talk to the OS and extensive cross-platform libraries, could revive Lisp. He is currently working on his own dialect, called Arc.

3 The Sigma Lisp Language

3.1 Design Philosophy

The design of the Sigma Lisp language is guided by six basic principles.

Assume a sufficiently smart programmer

Sigma Lisp is designed for very smart programmers, or at least programmers who know what they're doing. Most languages, especially mainstream ones such as Java, have protections built in to prevent mediocre programmers from doing too much damage. For an intelligent programmer, however, this can be very restrictive. Sigma is designed to be as free-form as possible, and to trust the programmer if he fools around with the interior workings.

Expressive enough to use and redefine itself

Virtually all of Sigma, including many operations normally thought of as "native", such as `quote`, and be expressed in pure Sigma. This means that the language can be reformed to fit its user's habits and style, virtually without limit.

The programmer's time is more important than the computer's

Today's more powerful machines mean that there is a real difference between "as fast as possible" and "fast enough", and there is room to trade efficiency for simplicity. It is pointless to optimize an operation if it won't make a noticeable difference, while wasting the programmer's time.

Language, then implementation

Sigma is not defined by this interpreter, or any eventual compiler. Sigma is, foremost, a language to express programs. At this point, I cannot worry about how something can be compiled, as long as it's possible and truly helps the programmer.

I can't do everything myself

Sigma cannot stand as an island, but needs to be able to interact with a variety of programming environments. Sigma needs to be easily extensible, in as many ways as possible. Furthermore, Sigma needs to be able to work with as many programming paradigms as possible.

Nothing is sacred

This is a new start, and any concepts from other languages, including other Lisps, will be examined solely on their merits. Everything about Lisp will be questioned to see if it really helps programmers.

3.2 Major Differences from Common Lisp

3.2.1 Execution

The plan for Sigma is currently to make only an interpreter, which can be accessed from a command line interface similar to Python. Common Lisp, depending on the implementation, can usually be interpreted through a command line or compiled into byte code or native code.

3.2.2 Namespace

Sigma is a Lisp_1, with a single namespace for variables and functions. This allows for more flexibility in manipulating expressions of functions. Common Lisp is a Lisp_2, with separate namespaces for variables and functions, which helps avoid name conflicts.

3.2.3 Macros

Macros in Sigma are substantially different from macros in Common Lisp. In Common Lisp, macros are not actual objects and are usually expanded at compile-time, which means that they cannot access runtime values, but execution speed is greatly improved. In Sigma, macros are first-class objects as functions are, and as Sigma is not compiled, all macros are expanded at run-time. They mimic the behavior of Common Lisp macros by preventing evaluation of their arguments, and the expression they return is evaluated to gain the final value of the macro call. As they are done at run-time, Sigma macros can access run-time values. This also means that they can recurse over values.

3.2.4 Function Applications

In Common Lisp, function application forms must begin with a symbol or a lambda form, largely due to its two namespaces. In Sigma, the first element can be any expression that returns a function or macro.

3.2.5 Special Forms

Common Lisp uses special forms for operations such as `quote`, which are keywords for the compiler. Sigma uses value macros, which are similar to macros in that their arguments are not evaluated, but their return value is treated as the return value of the entire macro application. Sigma also provides the ability to define value macros.

3.2.6 Backtick

In Common Lisp, backticked forms are expanded at read-time into an equivalent quoted form. In addition, the exact semantics of commas are implementation-specific. In Sigma, backticks serve as an abbreviation that the parser translates into an application of the macro `backtick`, and commas in turn simply tag their value with `&bt-eval`. The form is left unexpanded until the `backtick` application is evaluated.

3.2.7 Vectors

In Common Lisp, vectors have separate interfaces and semantics from linked lists made with `cons`. In Sigma, lists and arrays are interacted with through a shared interface, and arrays can, to an extent, emulate the behavior of lists.

3.3 Types

Nil

Nil serves as Sigma's nulltype. It is also used to indicate the end of a linked list.

Symbol

S-expressions use symbols to represent variable names. In addition, they are also commonly used in place of enumerated values. All symbols are stored in a registry, and the string representation of a symbol is unique among symbols. This allows for O(1) equality testing and the creation of a correct `gensym` function.

Cons

Cons cells are binary structures that store two values: `car` and `cdr`. A cons cell often represents a linked list where `car` stores the first element and `cdr` stores the remainder of the list, or nil if the cons stores the last element.

Number

Sigma uses a unified number type that can store a number as a native int, a native float, an arbitrary precision integer or a rational number.

String

Strings in Sigma are composed of four byte wide characters that store Unicode values.

Array

Sigma arrays are dynamically resizable, and are designed to be almost completely interchangeable with linked lists. In addition to using an identical interface, arrays can share their native data array with their subsections, which enables arrays to emulate some of the behavior of linked lists, as well as allowing subsections to be found in O(1) time.

Hash

A hash is a structure that maps strings or symbols to a value. These hashes draw no distinction between a string and a symbol with the same string representation.

Function

Functions have been first class objects in Lisp since it was originally defined, which allows for a range of operations whose flexibility could at best be clumsily simulated otherwise.

Macro

In Sigma, macros are first class objects as functions are. Their interface is identical to that of functions, and can be declared to return a value rather than an expression that is evaluated in its place.

Class

A class definition for Sigma's object system, these classes support multiple inheritance.

Instance

An instance of a class, instances can have methods defined that allow it to emulate objects of other types, such as lists or hashes.

Error

The error type is a general name for a range of objects that include exceptions, signals, and native signals as well as errors. Errors can either be inactive, when they can be manipulated normally, or active, in which case they interrupt evaluation and force it to return the error, causing the error to propagate upward until a `try` block or the toplevel is encountered.

4 Program Structure

4.1 Design Principles

The Sigma interpreter is a large, complex program that requires many different components to operate together perfectly. In addition, its implementation language, C, is an unforgiving language. As such, the program has been designed as much as possible to be easily tested and verified.

4.1.1 Functional Programming

In functional programming, functions are used primarily for their return values, and are expected to work using their specified parameters, without needing to check elements of program state such as global variables. In addition, functions ideally cause no side-effects. The advantage of this approach is that functions can be tested individually, working solely off of their arguments without needing to make a test suite to emulate a complete program state.

4.1.2 Bottom-up Design

Bottom-up Design emphasizes the building of tools by linking together smaller tools, ensuring that low-level data manipulation is not being performed except in controlled ways.

4.1.3 Synergy

Functional Programming and Bottom-up Design, when used together, allow code to be written easily, the completed code to be easier to understand, and for program components to be easily tested and verified.

4.2 Program Components

Sigma Lisp was programmed in separate components. Figures 1 and 2 in the appendix diagram the interactions of the components which will be explained in this section.

4.2.1 Basic Data Structures

The foundation of the Sigma interpreter is formed by basic structures for manipulating and storing data. Each structure, in addition to its basic definition, is accompanied by a series of functions that serve to carefully control interactions with the structures.

Hash A structure which maps Sigma strings to values. See figure 3.

Array A dynamically resizable array. Can share its data array with subsections, as to emulate linked lists. See figure 4.

Long An arbitrary precision integer.

Real A rational number formed using two **Long**s.

Registry Maps keys to the number of times they have been registered. Used to keep track of the usage of various data structures, such as symbols. See figure 5.

4.2.2 Sigma Data Structures

A number of data structures are built on top of the basic structures and are specific to Sigma.

Object Represents a Sigma data object, such as a number, string, or list. See figure 6.

Func A structure containing the definition of a function or macro. See figure 7.

Num A structure to allow a single interface for interactions between various types of numbers. See figure 8.

Cons A binary structure made of two cells, which is used to build linked lists as in Common Lisp. See figure 9.

4.2.3 Parser

The function **parse()** takes a native string as input and returns an **Object** representing the inputted S-expression.

4.2.4 Scope

Interactions with variable environments are controlled by a number of functions for creating branching, deleting, and storing values in variable environments as represented by **Object**s.

4.2.5 Eval

The function **eval()** and the accompanying function **apply()** form the heart of the Sigma interpreter. The **eval()** function takes an **Object** as returned from **parse()** and an **Object** representing the calling environment and evaluates the **Object** as an expression, performing any side effects, and returns the result. See figure 10.

4.2.6 Libraries

In order to do anything useful, a number of native functions will be needed to define basic functions such as `car` and control structures such as `while`. Many other functions and macros, such as `list`, will be defined in Sigma instead of native C.

4.2.7 Memory Management

Sigma uses a hybrid reference-counting and garbage collection system to manage memory. By keeping track of how references are being made to an **Object**, the interpreter can safely delete it when there are no references to it. The garbage collector is primarily a back-up system for handling circular references, as reference counting is generally faster, easier to use and far more predictable.

4.2.8 Interpreter

Once all the components are complete, a method for initializing them, linking the native and defined libraries into the toplevel, and an interface for the system will complete the interpreter. The interpreter will be, as much as possible, written in Sigma. This allows for easier modification of the system, as well as less work in verifying its functionality.

5 Problems and Issues

5.1 Speed

As it stands, the interpreter is very slow, as there is no compile-time during which to pre-compute any values, macros are expanded at run-time and so must build S-expressions on the fly, and most elements of the program were implemented in very inefficient ways. For example, variable environments are implemented by a linked list of `cons` cells with the layer level and a **Hash**. Every call to a global function requires recursing through multiple layers, with a hash lookup at each level.

5.2 Environments

There is currently no system for managing multiple namespaces, which restricts how large a program can realistically be. The planned solution was to devote a scope level to switching modules in and out, but this is a cumbersome solution at best, and does not allow for code across multiple files to be written without knowing in advance how it will be stored.

5.3 Garbage Collection

The current design of the Sigma interpreter precludes the possibility of doing effective garbage collection. The problem lies in that native functions such as **eval()** manipulate and create

Objects, such as the list of arguments to pass to **apply()**, without any way to recognize the function's link to them from outside the function. Furthermore, methods such as tagging **Object**s that are being manipulated by native code for the purpose of the garbage collector rely on being able to un-tag the **Object** later, which is unrealistic. As such, Sigma currently relies solely on reference counting.

6 Fixes and Future Plans

6.1 Symbols

One possible way to improve the speed of symbol lookups is to use a system of activation records and racks. An activation record is a record of a variable declaration, storing the symbol, the value, and the level. A rack is an array of stacks, each of which corresponds to a symbol. The value of a symbol can be obtained by looking at the top of its stack. Forms such as `let` and `fn` which declare variables push the new value onto each stack, and when the form closes, they are popped off, restoring the previous value.

6.2 Continuations

Currently, Sigma does not have any form of explicit continuations, although functions like `call/cc` can be emulated with errors. However, the inclusion of continuations would solve several problems with Sigma, as well as increasing its expressive power.

6.2.1 The Concept of Continuations

The idea of continuations is to isolate the information needed to evaluate an S-expression. Obviously, one needs the expression itself, as well as the variable environment to perform symbol lookups in. A third piece of information is where the value is expected to go, called the continuation. For example, in the expression `(car (cons a b))`, the continuation of the expression `(cons a b)` can be represented by `(car [])`, or alternatively, a function: `(fn (x) (car x))`.

6.2.2 Continuations in Scheme

Continuations in Scheme are manipulated by the function `call/cc`, which is short for `call-with-current-continuation`. `call/cc` is called with one argument, a function that is also of one argument, as so: `(call/cc (fn (k) ... ))`. When `call/cc` is called, it calls its function argument with the continuation the form was meant to return to, reified as a unary function. For example, in the expression `(print (call/cc (fn (k) ... )))`, the function `(fn (k) ... )` would be applied to the continuation `(print [])`. Within the function body, `k` would be a function that when called, would return its argument as the result of the entire `call/cc` form. If `k` is never called, the value returned from the function is returned from the `call/cc` form as well.

6.2.3 Continuations in Sigma

An expansion of this idea could be used to implement continuations in Sigma, as well as fixing a number of difficulties in Sigma. If the idea of a continuation is expanded from a computation expecting a value to a template for a computation, expecting a number of values, continuations can be used to control the flow of the program. With this model, a continuation draws from a source and places values into a template for a function application, and would store the variable environment and the continuation it will return to. For example, if a continuation is represented by `(func args src env k)`, then the continuation for the form `(cons a b)` that returns to k_0 in the environment env_0 would be `(NULL [NULL NULL] (cons a b)` env_0 k_0 `)`. The program would run by 'prompting' the current continuation, which would draw each element out of its source and evaluate it. A symbol or atom would be evaluated as normal, but a function application would reassign the current continuation to one representing the form whose continuation would be the former current continuation.

7 Conclusion

Our goal in creating this language and its interpreter is to create as expressive a language as possible. Lisp offers more capability for redefining its abstractions, semantics, and constructs than any other language, and Sigma offers a greater degree of flexibility than other dialects. While fleshing out the language's libraries enough to be easy enough to use for large projects is beyond the scope of this project, a flexible framework is in place to allow for expansion.

References

[1] Paul Graham. *On Lisp*. Prentice Hall, 1993. ¡http://www.paulgraham.com/onlisptext.html¿

[2] Paul Graham. *Being Popular*. May 2001. ¡http://www.paulgraham.com/popular.html¿

[3] Paul Graham. *Five Questions about Language Design*. May 2001. ¡http://www.paulgraham.com/langdes.html¿

[4] Paul Graham. *The Roots of Lisp*. May 2001. ¡http://www.paulgraham.com/rootsoflisp.html¿

[5] Brian Harvey. *Symbolic Programming vs. the A.P. Curriculum*. 30 May 2003. ¡http://www.cs.berkeley.edu/ bh/bridge.html¿

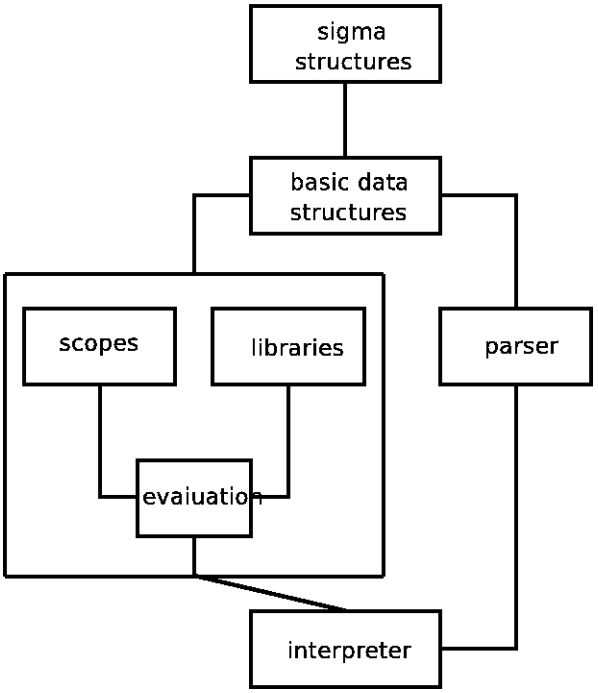
[6] John McCarthy. *Recursive Functions of Symbolic Expressions and their Computation by Machine (Part 1)*. 19 November 2003. ¡http://www-formal.stanford.edu/jmc/recursive.html¿

[7] Pascal Costanza. *Pascal Costanza's Highly Opinionated Guide to Lisp*. 26 August 2005. ¡http://p-cos.net/lisp/guide.html¿

[8] Rodney Brooks and Richard Gabriel. *A Critique of Common Lisp*. 1984. ¡http://www.dreamsongs.com/NewFiles/clcrit.pdf¿

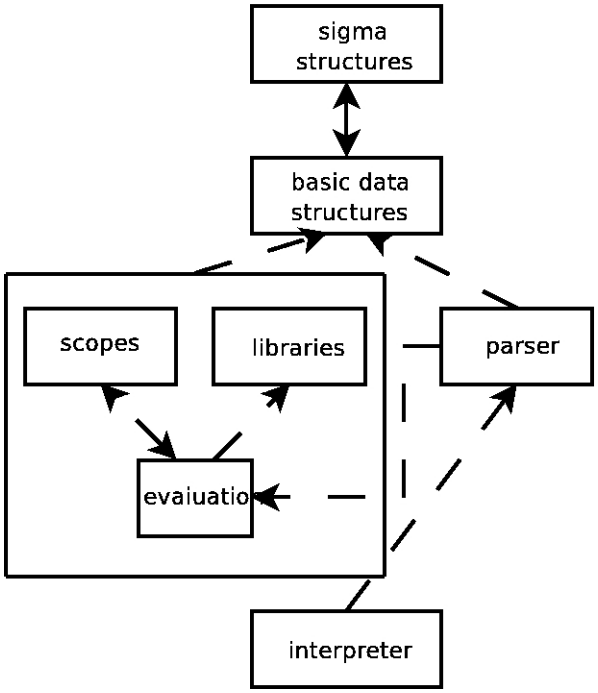
[9] Dave Marshall. *Programming in C*. March 1999. ¡http://www.cs.cf.ac.uk/Dave/C/¿

8 Appendix



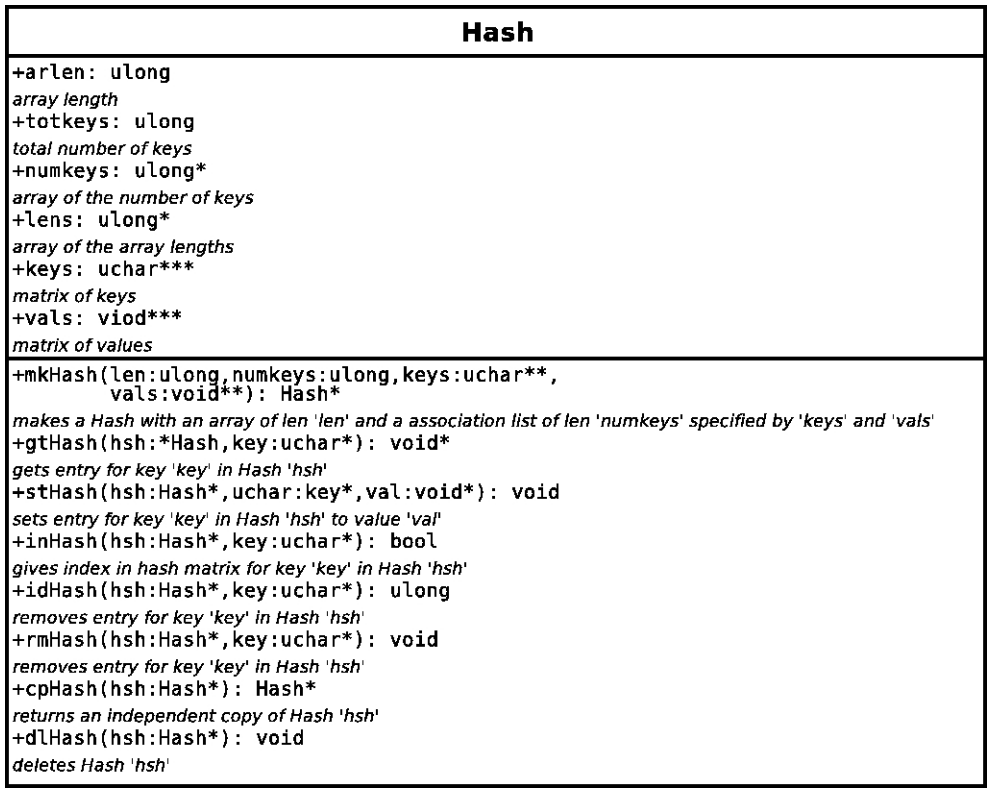
Lisp/Diagrams/Sigma Lisp Dependency Diagram.jpg

Figure 1: A diagram showing how the components of Sigma Lisp are connected



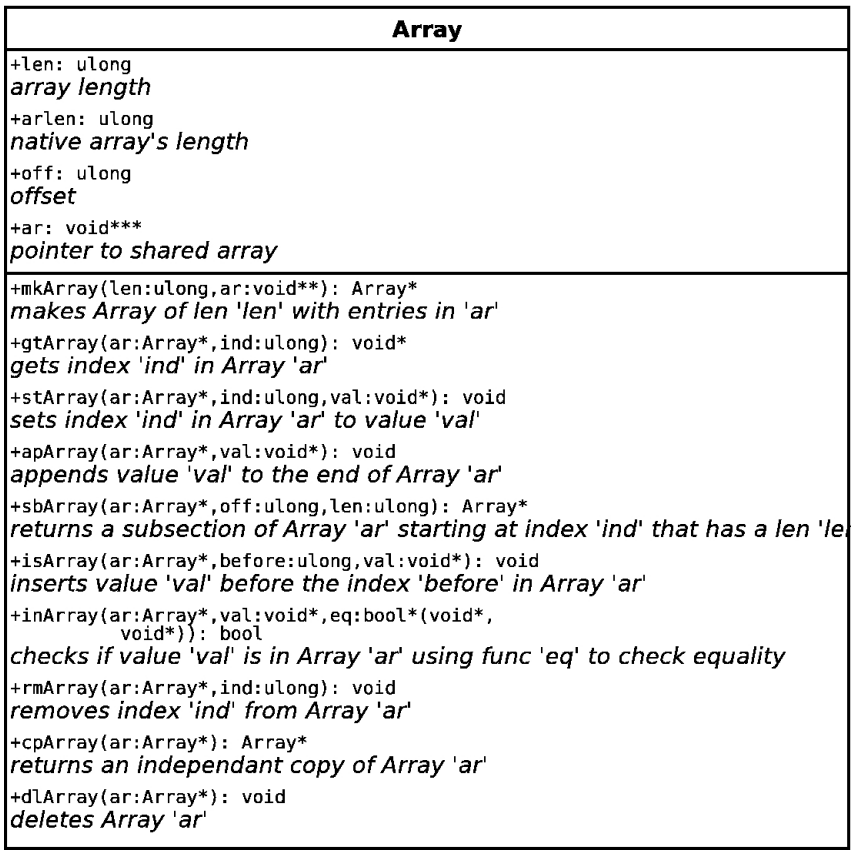
Lisp/Diagrams/Sigma Lisp Data Flow Diagram.jpg

Figure 2: A diagram showing how data travels between the components of Sigma Lisp



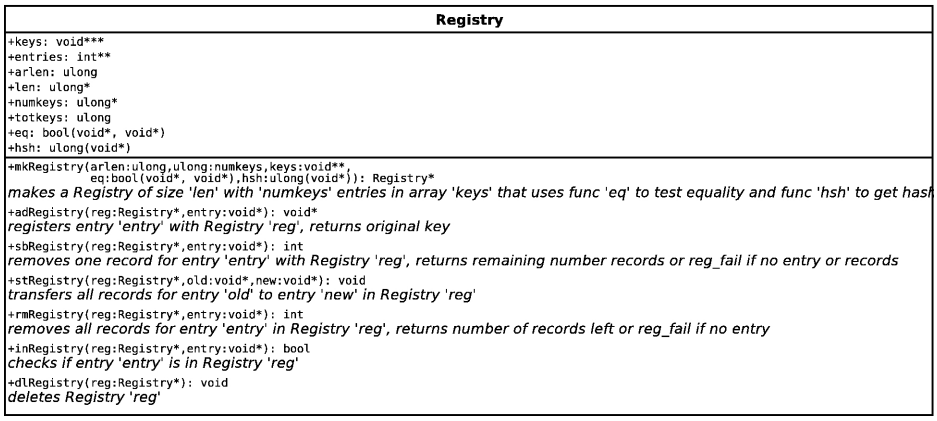
Lisp/Diagrams/Hash.jpg

Figure 3: A diagram of the Hash data structure



Lisp/Diagrams/Array.jpg

Figure 4: A diagram of the Array data structure



Lisp/Diagrams/Registry.jpg

Figure 5: A diagram of the Registry data structure

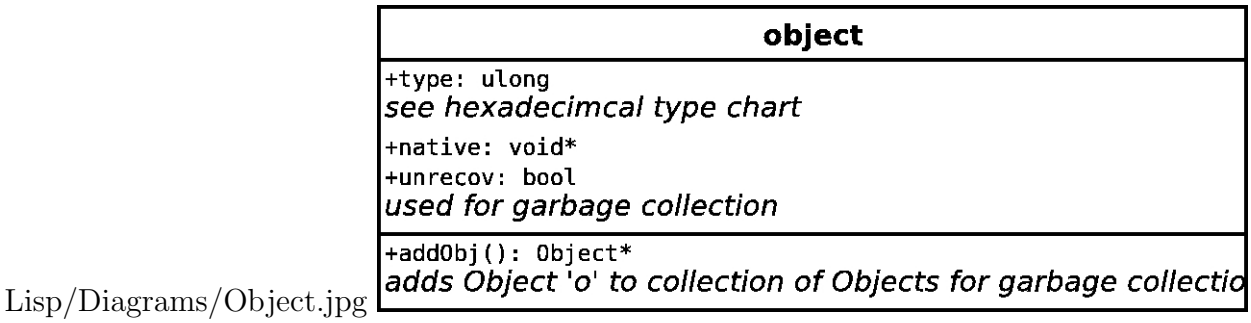


Figure 6: A diagram of the Object data structure

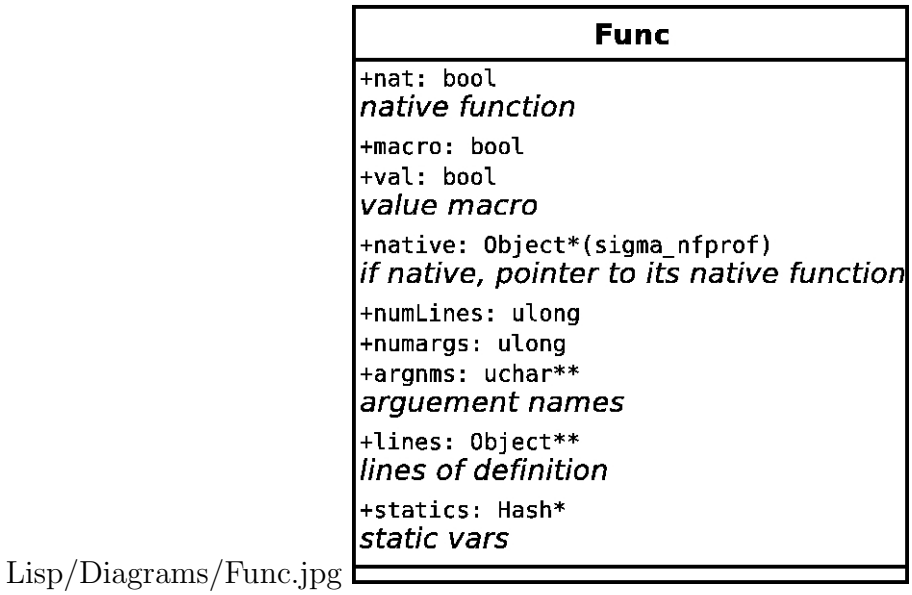


Figure 7: A diagram of the Function data structure

Lisp/Diagrams/Num.jpg

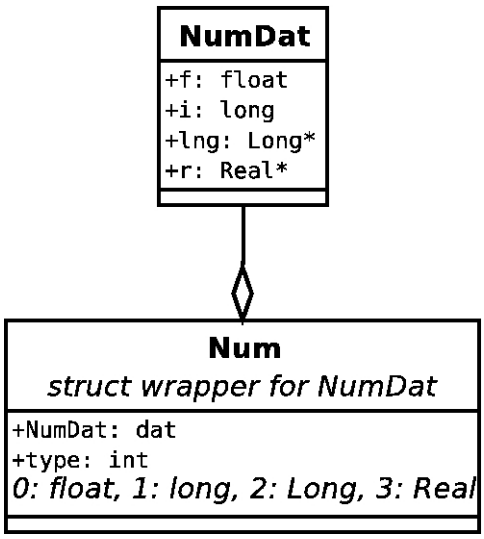


Figure 8: A diagram of the Number data structure

Lisp/Diagrams/Cons.jpg

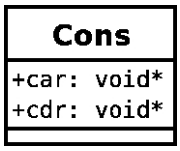


Figure 9: A diagram of the Cons data structure

Lisp/Diagrams/EvalApplyTrace.jpg

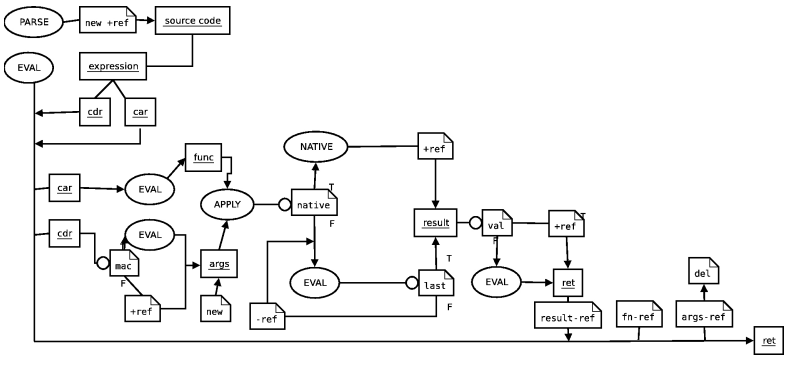


Figure 10: A diagram of the process performed in the eval() and apply() functions, with additional attention to tracking reference counts