The Design and Implementation of a Modern Lisp Dialect

Sam Davis

Nicholas Alexander

January 26, 2006

Abstract

Lisp, invented in 1958 by John McCarthy, revolutionized how programs could be written and expressed. Instead of giving explicit instructions to a computer, Lisp expressed programs as logical operations and functions. Lisp was the first language to incorporate modern language features including: garbage collection, conditionals, first class functions, and recursion. Also, in 1994 Common Lisp became the first language to use object oriented programming. However the emergence of the popular C model of programming has shifted the perception of what level of detail (high or low) a programming language ought to focus on. Today programming languages are meant to interact with the operating system at an intimate level and communicate with other language environments. Common Lisp, which focuses on higher level details, has become, accordingly, outdated and considered ineffective. However, the Lisp model of

programming still arguably remains the most effective for creative applications. Lisp has a greater ability to abstract and works at a higher level than C code. Therefore a fundamental difference arises in the style of program development between the two languages. This paper will attempt to explain the benefits of programming from the Lisp side.

1 Introduction

1.1 Purpose

Brian Harvey writes in his essay Symbolic Programming vs. the A.P. Curriculum that today the primary focus on programming is the software engineering approach. That is, programs are taught and expected to start "with the broad ideas...and fill in the details later" (Harvey). However, software engineers tend to "focus on low-level details [such as] explicit storage allocation... strong typing...[and] hardwired control structures"(Harvey). Furthermore "the transition from C (a hacker's language) to C++ (a software engineer's language) has brought more attention, not less, to...low-level details." Therefore, this project intends to update Lisp's abstraction and higher level approach to match the popularity of C++'s low-level detail-oriented coding. Harvey notes the proposed abstract approach "can focus on ... issues [such as] functional programming, object-oriented programming, and logic programming methodologies, data abstraction, and higher levels of abstract such as the design and implementation of a programming language" (Harvey). Sigma Lisp therefore intends to give competent programmers with "artistic" intent the tools to make new and creative

developments. Indeed, Harvey notes "a Lisp procedure written in terms of subprocedures that may not yet be defined provides the right level of abstraction, while retaining technical rigor" and therefore is the ideal tool to promote creative programming. Naturally, this project is not meant to rival the C model of programming (which should be used where lives are threatened, as Harvey states) but instead provide an alternative toolbox for the programmer who desires to create something revolutionary and no less important but far less risky. Our contention is, when lives are not involved, low-level errors are an acceptable trade-off for high-level abstraction capability.

1.2 Scope of Study

An interpreter, not compiler, that can accept explicit prompts or file inputs and execute them.

2 Background and Review of Literature

Instrumental to the concept of this project were Paul Graham's essays on effective language design and McCarthy's original paper on Lisp. One of the guiding philosophies for Sigma Lisp is borrowed directly from Graham: "...almost anything you can do to make programs shorter is good. There should be lots of library functions; anything that can be implicit should be; the syntax should be terse to a fault; even the names of things should be short" (Graham, Five Questions About Language Design). Essentially we hope to create a programming language designed for hackers (elite programmers) with a high level of abstraction, something Graham

highly praises.

2.0.1 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.

2.1 Literature

2.1.1 Background of Lisp

McCarthy's paper, however, is used more to gain an understanding of the fundamental Lisp language and theory, a concept which won't be explained in too much detail here. However, since we plan upgrade Lisp and maintain most of the original concepts proposed by McCarthy the paper was fundamental in enhancing our understanding of what Lisp means. Lisp is based on S-expressions, symbolic expressions translatable by machine into significant data. More

specifically, S-expressions are derived by extending lambda calculus by adding conditions as a way to express code as a list. Essentially the concept of applying calculus and algebra to a computer language. In fact, McCarthy became interested in Lisp primarily as a means for writing computer derivation software.

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

3.3 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.

Method

A method is a function or macro that has been called by an instance, allowing variables from the caller to be modified easily.

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

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.

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

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.

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.

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

Scope Represents an environment mapping variables to values.

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

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 **Scope**s are controlled by a number of functions for creating branching, deleting, and storing values in variable environments as represented by **Scope**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 a **Scope** representing the calling environment and evaluates the **Object** as an expression, performing any side effects, and returns the result.

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.