

# Obsidian Level-0 Interfacing

March 28, 2009

## 1 Preface

Why another programming language? In my mind, because all of the great ideas are spread out among too many disconnected technologies. Haskell’s purity, Lisp’s macros, and Unlambda’s simplicity are all noteworthy, but I have to choose which of these features I want to use. Arguably I shouldn’t have to do this; there are ways of integrating these ideas.

One of the best ways to integrate ideas is to make everything as simple as possible. Then features are written within the language and not inside the interpreter. This great idea, naturally, was embodied by Lisp and Scheme; it requires that your code makes sense as a data structure as well as being executable. One might argue that Haskell provides the equivalent, but it really doesn’t; for instance, you cannot build operator precedence into the language without changing the compiler. (To be completed later)

Incidentally, one of the conventions I’ll try to stick to is to represent things that are best considered transformations or functions at an intuitive level as either Greek letters or upper-case Roman ones, and arbitrary values will be represented by lower-case Roman letters. Constants are generally represented by mathematical symbols of some sort.

## 2 Introduction

The level-0 interfacing is all provided by the runtime. Rather than standardizing a particular library implementation, this is considered to standardize the interpreter/compiler itself.

### 2.1 Parsing

The grammar is remarkably simple. Here is the complete form, using regular expressions and no “magic”:

```
<symbol> ::= /^[^s()]+/  
<list>   ::= /\({<symbol>|<list>}*\) /
```

## 2.2 List representation

When the reader inputs a list, it is stored in the opposite order as the equivalent list in Lisp or Scheme. For instance, the list (1 2 3) is stored as (((() . 1) . 2) . 3), where we refer to 3 as the head of the list and (((() . 1) . 2) as the tail. While initially counterintuitive, this representation reflects the evaluation rules of the language. In expressions, I often represent the cons pair (x . y) as  $x :: y$ .

## 3 Evaluation

A cons cell  $x :: y$  is evaluated as  $xy$ . In other words, the tail of the cons cell is invoked on the head. If either is another cons cell, then that cons cell is placed inside a lazy evaluation thunk and then used. Thus cons cells act as a mechanism for enforcing the order of operations.

Below are the exact semantics of the level-0 functions provided by the interpreter. For convenience, we assume the presence of the function  $I = SKK$ , though the interpreter is not required to provide it (and must be assumed not to).

$$()x = x \tag{1}$$

$$S\psi\pi x = \psi x (\pi x) \tag{2}$$

$$Kxy = x \tag{3}$$

$$Q\phi_1\phi_2\omega x = \begin{cases} \omega(\phi_2(Q\phi_1\phi_2Ia)(Q\phi_1\phi_2Ib)) & \text{if } x = a :: b \\ \omega() & \text{if } x \text{ is the end of a list} \\ \omega(\phi_1x) & \text{otherwise} \end{cases} \tag{4}$$

$$xy = \begin{cases} I & \text{if } x \text{ is an unbound symbol} \\ xy & \text{otherwise} \end{cases} \tag{5}$$

Note that in the case of the  $Q$  combinator I am cheating with the notation a bit. The purpose of having  $Q$  in the language is that you are allowed to quote certain expressions within the source code to ensure that they are not evaluated at all, but rather treated as data that you happened to type in. This is a prerequisite for true syntactic macros. Here are some examples of what  $Q$  should produce:

$$\begin{aligned} Q I K I (a b c d e f) &= () \\ Q I cons I (a b c d e f) &= ((((((().a).b).c).d).e).f) \\ Q (K Q) cons I (a b c d) &= ((((((.Q).Q).Q).Q) \end{aligned}$$

The major point here is that  $Q$  takes a literal expression, no matter what that expression means, and lets you construct a more useful representation of it in code. The structure it gives you is never evaluated, and you will see it as a tree of symbols.

## 4 Constants

In order to save typing and redundant evaluation cycles, an odd evaluation rule is provided (rule 5 in the list above). Its function is to store a value for later use. After invoking an unbound symbol on a value, the interpreter will replace instances of the symbol with the value. Naturally, this substitution does not affect the value of expressions quoted with the  $Q$  combinator.

Having implicitly-defined constants such as this serves a dual purpose. First, it simplifies the base of the language by not introducing another function; and second, it serves to make typographical errors impossible to detect. In my mind, the first justifies the second.

I realize that it is possible to implement constants as a corollary of the fact that we have combinators; that is, we may pass a value into a function and let that serve as a means to implement constants. However, this is preposterous for real-world programming (it saves little typing) and would be a sadistic gesture to users of the language.