HLisp Specification

Taran Lynn

May 3, 2016

Abstract

HLisp is an attempt to make a minimal lisp with a very small core. All computations are done through list and symbol processing, and numeric support is not built-in. IO is also unsupported, as everything is expected to be done from the REPL. As such HLisp is obviously designed for ease of implementation and academic research.

Contents

1 Data Types

HLisp supports four different primitive data types, which include

Cons Cells Cons cells are pairs of data. They have two slots which can hold any other lisp datum. If `a` and `b` are data, then a cons cell is denoted as `(a . b)`.

Functions Functions should fulfill several requirements.

1. Functions should eagerly evaluate their arguments.
2. All functions should be lexically scoped.

Macros Macros are data just like functions, except that their arguments are passed unevaluated. That is, if the symbol `a` is passed as an argument, then the macro will bind the symbol `a` to the appropriate variable, and not bind whatever the symbol `a` is bound to in the current environment.

Nil Nil is a datum that represents nothing, the end of a list, or boolean false. This paper denotes it as ().

Symbols Symbols are named identifiers. They consist of any character the isn't whitespace or '(', ')', ';', or '.'.

There are two things that should be noted about HLisp's data. First, all data is immutable. Second, all data that isn't nil represents boolean true, and nil represents boolean false.

In addition to these basic data types there is a compound type, which is the list. A list is either the nil datum or a cons cell whose second slot contains a list. A list is denoted by a group of space separated data within parenthesis.

```
list = (datum . list)
     | ()
```

2 Syntax and Semantics

2.1 EBNF Grammar

```
datum = list | quote | symbol ;
```

```
data = { datum } ;

list = '(', data, ')' ;

quote = "'", datum ;

comment = ';', { nonnewline } ;

symbol = { nonspecial } ;

nonspecial = ? any character except whitespace or '(', ')', ''', and '.'. ? ;

nonnewline = ? any character that doesn't create a new line ? ;
```

2.2 Evaluation

There are a few evaluation rules in HLisp, mainly

1. All quotes `'a` are expanded to `(quote a)`.
2. Symbols are looked up in the environment for a datum value.
3. For a list `(f ...)`, if `f` corresponds to the name of a special form, than an action specific to that special form is performed. If `f` is a macro or function, then it is applied to the arguments. Otherwise, `f` is evaluated and the process is repeated.
4. Function arguments are evaluated and then passed to the function.
5. Macro arguments are passed unevaluated, but the return value of the macro is evaluated.

2.2.1 More on Macro Evaluation

Unlike in most lisps macros can be passed around as data (though special forms can't be). If we pass a macro to a function, and that macro is invoked, then the result will be the same as if we replaced the variable with the macro's bounded name. For example, consider the following expansion.

```
((lambda (m a) (m (a 'b) a)) let 'a)
=> ((lambda (a) (let (a 'b) a)) 'a)
=> ((lambda (a) ((lambda (a) a) 'b)) 'a)
=> b
```

This could not be done if `f` was passed as a normal function. This is a trait unique to HLisp (to the best of my knowledge).

3 Primitive Functions

The following are primitive functions defined in the run-time implementation.

(cons x y) Creates a cons cell whose first slot is occupied by `x` and second is occupied by `y`.

(car xs) Returns the datum in the first slot of a cons cell.

(cdr xs) Returns the datum in the second slot of a cons cell.

(= x y) Returns the symbol `t` if the two objects are equal, otherwise it returns nil.

(apply f xs) Applies the function `f` to the list `xs`.

(atom? x) Returns the symbol `t` is `x` is not a cons cell, otherwise nil is returned.

(eval x) Evaluates `x` in the context of the current environment.

4 Special Forms

The following are special forms, which are evaluated according to special rules.

(if pred x y) Evaluates `pred` and does one of two things based on its value. If it is nil, then `y` is evaluated and returned. Otherwise, then `x` is evaluated and returned

(label name x) Binds the symbol `name` to `x` so that when it's evaluated it returns `x`. Binding is restricted to local scope within function definition. So `(lambda (x) (label y x)) 'a` will **not** create a global variable `y`.

(lambda (args...) body) or (lambda args body) Creates a new function. When evaluated this function evaluates `body` with the variable symbols rebound to the argument's values. Function arguments are either a list of symbols to be bound to positional argument, or a single symbol bound to a list of the arguments. This form should also handle closures.

(macro (args...) body) or (macro args body) Works just like `lambda`, except that it creates a macro.

(quote x) Returns its argument `x` unevaluated.