Implementing Algebraic Data Types, Lazy Evaluation, And Monads In Common Lisp

Slava Akhmechet
coffeemug@gmail.com

Abstract

In this paper I present *cl-monad* - a library that introduces algebraic data types, lazy evaluation, infinite data structures, and monads to Common Lisp. I develop a notation that elegantly integrates these features into Common Lisp and discuss its compilation into Common Lisp code. Additionally, I implement common Haskell constructs `Maybe` and `List` along with supporting utility functions. Finally, I solve famous problems (such as infinite Fibonacci sequences with on demand generation in linear time, generation of a powerset, and solving non-deterministic equations) using the library.

General Terms Abstract Data Types, Monads, Lazyness

Keywords ADTs, Monads, Lazyness, Lisp

1. Introduction

In recent years much of the academic programming language research as well as commercial development based on functional languages has centered around Haskell. Some of the attractive Haskell features include algebraic data types, delayed evaluation, and monadic abstraction. These features allow esoteric solutions to commonly encountered problems. In particular, delayed evaluation allows construction of infinite data structures that facilitate elegant generation of infinite sequences such as Fibonacci numbers. Combination of algebraic data types and monads allows elegant implementation of many desirable abstractions including `maybe` type and non-determinism. In this paper I show that Common Lisp is sufficiently powerful to elegantly implement these concepts. I provide a reference implementation via a Common Lisp library *cl-monad*, and use the developed framework to create infinite Fibonacci sequences with on demand generation in linear time, generate a powerset using monadic abstraction and `do`-notation, and solve non-deterministic equations.

2. Algebraic Data Types

In this section I explore the problem of adding ADTs to Common Lisp. I first consider how to express ADTs in terms of Common Lisp primitives. Once appropriate representation is determined, I present a compiler that converts custom ADT notation to Common Lisp code. I then customize Lisp printer to make ADT printing similar to that of Haskell and introduce a pattern matching facility. At the end of the section commonly used algebraic types `Maybe` and `List` become available in Common Lisp.

2.1 Expressing ADTs In Common Lisp

Expressing ADTs in Common Lisp requires solving two problems: defining a notation for declaring ADTs in Lisp code, and finding an appropriate Lisp representation this notation can be compiled into.

2.1.1 ADT Notation

The standard `Maybe` type is defined in Haskell as follows[4]:

```
data Maybe a = Nothing | Just a
```

First, the data type is specified, then constructors and types of their fields are listed, separated by vertical bars. Ideally a Common Lisp notation for declaring ADTs would be similar. A notation I chose uses a symbol `defalgebraic` to declare an ADT, followed by a type name and a sequence of lists, where each list begins with a constructor name followed with type specifiers of their fields[1], if any. The type `Maybe` can be translated to this notation as follows:

```
(defalgebraic maybe (nothing) (just t))
```

For convenience, parentheses may be omitted if the constructor has no fields and type specifiers for any type `t` may be replaced with `*`:

```
(defalgebraic maybe nothing (just *))
```

Additionally, this notation must allow the definition of data types recursively. Consider standard Haskell definition of a list[4]:

```
data List a = Nil | Cons a (List a)
```

This definition can be translated to `defalgebraic` notation as follows:

```
(defalgebraic @list @nil (@cons * @list))
```

I use the character `@` to differentiate a *cl-monad* list from the standard Common Lisp equivalent.

In Haskell an instance of an abstract data type may be constructed as follows[4]:

```
let x = Just 42
```

An empty constructor is used without any arguments[4]:

```
let y = Nothing
```

These expressions can be translated to Lisp notation as follows:

```
(setq x (just 42))
(setq y nothing)
```

[1] Note that the notation adapts type specification to that of Common Lisp, and does not attempt to replicate Haskell's typesystem-related functionality.

2.1.2 ADT Representation

In order to smoothly integrate ADTs into Common Lisp, an appropriate representation for the above notation needs to be defined. Common Lisp Object System provides a convenient medium for expressing ADTs[8]. I define a root class for all algebraic types that can be used to customize their behavior:

```
(defclass algebraic () ())
```

I then derive any given type from `algebraic` and use it as a parent type for constructors. This allows the types to maintain an *is-a* relationship: any given type is an `algebraic`, and any given constructor is of that type.

Additionally, I use slots to represent constructor arguments. Each slot's type specifier encodes and verifies the type information obtained from the ADT notation.

One representative feature of ADTs is that data may be extracted from their instance only via pattern matching. I use `gensym` to generate slot names in order to avoid direct access. A matching mechanism is discussed in section 2.4.

Recall the definition of the `maybe` type:

```
(defalgebraic maybe nothing (just *))
```

It can be translated to Common Lisp representation as follows:

```
(defclass maybe (algebraic) ())
(defclass nothing (maybe) ())
(defclass just (maybe)
  ((#:g1234 :type t)))
```

Additionally, a function for instantiating the `just` constructor and the symbol for the instance of `nothing` need to be defined:

```
(defun just (a0)
  ;; instantiate class JUST
  )
(setf nothing ;; instantiate class NOTHING
  )
```

2.2 ADT Compiler

An ADT compiler hooks into Common Lisp by defining a macro `defalgebraic`. The macro accepts the notation presented in the previous section and compiles it into Lisp code. The basic compilation process involves generating code that performs the following steps:

1. Define a CLOS class for the algebraic type
2. For each constructor:
   (a) Define a CLOS class for the constructor type
      i. Convert fields into CLOS slots
   (b) Define a function to instantiate the type via the constructor
      i. Typecheck the arguments
      ii. Instantiate appropriate class and set the slot values

In section 2.4 I extend this process to generate a custom pattern matcher for each constructor. In section 3.1 I modify the construction instantiation step to support lazy evaluation.

The implementation of `defalgebraic` is fairly straight forward [5, 1] and can be obtained as part of the *cl-monad* library.

2.3 Customizing Lisp Printer

Once the compiler is in place, ADTs can be defined and the constructors can be used to create their instances:

```
> (defalgebraic maybe nothing (just *))
=> MAYBE
> (setq x (just 42))
=> #<JUST {CE76571}>
```

The default representation the Common Lisp printer uses for CLOS objects isn't useful for ADTs. A much more useful behavior can be achieved by specializing the generic function `print-object`[7] on class `algebraic`. The implementation provided in *cl-monad* produces the following output:

```
> (setq x (just 42))
=> #A(JUST 42)
> (setq y (just (just 5)))
=> #A(JUST (JUST 5))
```

The `#A` notation indicates that the type is algebraic. The rest of the mechanism is very similar to Haskell: the constructor's name is printed, followed by the values of the fields.

While convenient for most algebraic types, this mechanism is too verbose for lists:

```
> (defalgebraic @list @nil (@cons * @list))
=> @LIST
> (@cons 1 (@cons 2 (@cons 3 @nil)))
=> #A(@CONS 1 (@CONS 2 (@CONS 3 @NIL)))
```

To alleviate this problem *cl-monad* customizes the printer by specializing `print-object` on `@list`, and introduces a utility function `@list` that allows a more comfortable list creation:

```
> (@list 1 2 3)
=> #A[1 2 3]
```

2.4 Pattern Matching

An open source library `fare-matcher`[6] provides excellent building blocks for developing ADT pattern matching. I modified the compiler discussed in section 2.2 to generate a compatible matcher when constructors are compiled. The core of custom matcher generation is defined in `define-algebraic-matcher` function (available in appendix A). This function generates a matcher that compares an expression to a pattern, ensuring that constructor names and argument arity matches, at which point it returns values stored in the data instance. The compiler calls `define-algebraic-matcher` to generate a matcher for each constructor.

The `maybe` type can now be recompiled in order to generate an appropriate pattern matcher. Pattern matching can be performed via `match` - `fare-matcher`'s equivalent of Haskell's `case`:

```
> (defalgebraic maybe nothing (just *))
=> MAYBE
> (match (just 42)
    (nothing nil)
    ((just x) x))
=> 42
```

Similarly, pattern matching can be applied to lists:

```
> (defalgebraic @list @nil (@cons * @list))
=> @LIST
> (match (@list 1 2 3 4 5)
    (@nil nil)
    ((@cons x xs) xs))
=> #A[2 3 4 5]
```

In addition, I extended `fare-matcher` with macros for matching multiple expressions:

```
> (mmatch ((@list 1 2 3 4 5) @nil)
    ((@nil @nil) nil)
    ((((@cons x xs) @nil) xs))
=> #A[2 3 4 5]
```

Based on this infrastructure, common utility functions can be defined in a style very similar to that of Haskell. Consider a definition of `@map` - a function for mapping ADT-based lists:

```
(defun @map (fn list)
  (match list
   (@nil @nil)
   ((@cons x xs) (@cons (funcall fn x)
                        (@map fn xs)))))
```

```
> (@map #'1+ (@list 1 2 3 4))
=> #A[2 3 4 5]
```

I will define more utility functions in section 3.2.

3. Lazyness

Much of Haskell's innovation comes from its lazy semantics. Lazy evalution allows for elegant solutions to familiar problems and optimizations. In this section I extend the ADT framework with lazyness and introduce infinite lists.

3.1 Extending ADTs With Lazyness

Extending ADTs with lazy semantics involves modifying the constructors to lazily evaluate their arguments. Because any expression may be passed as an argument to a constructor, in order to implement lazyness it's necessary to modify the compiler discussed in section 2.2 to represent constructors as macros instead of functions. Once constructor functionality is implemented via macros, the generated code gains the flexibility necessary to delay argument evalution. Constructor arguments can be wrapped into a classic `delay`/`force` mechanism where λ-functions are used to delay evaluation.

The remaining question is at which point to force evaluation. One option is to require the user of the library to force evaluation explicitly. This option gives the most control but is not in the spirit of Haskell and makes many of Haskell's features implemented in *cl-monad* significantly less useful. Another option - the one I implement in *cl-monad*, is to force evaluation at the time of pattern matching. Because data can only be retrieved from ADTs via a successful pattern match, this approach enables an automatic system that follows Haskell's semantics. It ensures expressions wrapped in ADTs are evaluated if only if needed.

Unfortunately, introduction of lazy semantics makes it impossible to perform type checking within the constructor. Due to lack of type inference the type of any given expression cannot be known until the expression is evaluated. Therefore, it is necessary to move type checking code into `force`, and because of lazy semantics the user may not find out about type errors at all! In this respect *cl-monad* fully preserves the spirit of Common Lisp.

3.2 Lazy Lists Utilities

In order to take advantage of lazy lists some commonly used utility functions[4] need to be implemented. Figure 1 presents some of the functions implemented in *cl-monad*: `@car` and `@cdr` return head and tail of the list, respectively. The function `@zip-with` accepts a function and two lists and applies the function to successive pairs of elements, collecting the results. The function `@take` returns the first `n` elements from a list.

Note that these functions automatically take advantage of lazy semantics by using pattern matching - the values passed to the constructor will not be evaluated until there is a successful match. This framework can be used to implement more esoteric features from Haskell - infinite lists.

```
(defun @car (list)
  (ematch list
          (@nil @nil)
          ((@cons a *) a)))

(defun @cdr (list)
  (ematch list
          (@nil @nil)
          ((@cons * b) b)))

(defun @zip-with (fn list1 list2)
  (emmatch (list1 list2)
           ((@nil *) @nil)
           ((* @nil) @nil)
           (((@cons x xs) (@cons y ys))
               (@cons (funcall fn x y)
                      (@zip-with fn xs ys)))))

(defun @take (n list)
  (if (<= n 0)
      @nil
      (ematch list
              (@nil @nil)
              ((@cons x xs) (@cons
                             x (@take (- n 1)
                                      xs))))))
```

Figure 1. Commonly used utility functions for list manipulation.

3.3 Infinite Lists - First Attempt

The simplest infinite list is a list of all integers. Such a list can built using a recursive definition: start with zero and increment each successive element by one:

```
(defun @integers (&optional (i 0))
  (@cons i (@integers (1+ i))))
```

It is now possible to take elements from the infinite list of integers:

```
> (@take 3 (@integers))
=> #A[0 1 2]
> (@take 10 (@integers))
=> #A[0 1 2 3 4 5 6 7 8 9]
```

Calling `@integers` without `@take` will result in a stack overflow due to an attempt at infinite thunking:

```
> (@integers)
=> Stack Overflow!
```

It's also possible to perform operations on infinite lists. For example, two infinite lists can be zipped into one:

```
> (@take 5 (@zip-with #'+ (@integers) (@integers)))
=> #A[0 2 4 6 8]
```

Every time the function `@integers` is called it generates a new list. This code can be rewritten in a different way to bind infinite lists to symbols:

```
> (defvar *@integers*
          (@cons 1 (@map #'1+ *@integers*)))
=> *@integers*
> (@take 10 *@integers*)
=> ...
```

The result couldn't be obtained - evaluation of the expression exhausts the stack. Upon closer examination it is clear that the pattern

matching mechanism built in `define-algebraic-matcher` is far too eager. Consider the definition of `@map`:

```
(defun @map (fn list)
  (match list
   (@nil @nil)
   ((@cons x xs) (@cons (funcall fn x)
                        (@map fn xs)))))
```

If the pattern matcher matches `list` against `(@cons x xs)`, both `x` and `xs` are immediately evaluated, even though the value of `xs` will not be needed until `@map` calls itself recursively within a new instance of `@cons` (which may never happen if no one evaluates its `@cdr`). Before the library can allow to build such lists this issue needs to be fixed.

3.4 Fixing Pattern Matching Eagerness

The fix for unnecessary eagerness is rather simple. Instead of forcing evaluation on pattern matching, the matching macro needs to transform the body of successful matches to wrap usage of relevant variables with a call to `force`. This ensures the expression is only evaluated if and when it is needed.

With this change in place it's possible to define Haskell-style infinite lists:

```
> (defvar *@integers*
          (@cons 1 (@map #'1+ *@integers*)))
=> *@integers*
> (@take 10 *@integers*)
=> #A[0 1 2 3 4 5 6 7 8 9]
```

In a similar manner *cl-monad* can be used to generate an infinite fibonacci sequence[2]:

```
> (defvar *@fib* (@cons 1 (@cons 1
                   (@zip-with #'+ *@fib*
                                  (@cdr *@fib*)))))
=> *@fib*
> (@take 10 *@fib*)
=> #A[1 1 2 3 5 8 13 21 34 55]
```

Taking more values from `*@fib*` reveals another issue with the library - the sequence is generated in exponential time. Because the evaluation machinery does not perform zero-arity functions memoization, a standard Haskell feature, every time a value from the list is needed, all preceding list items are reevaluated.

3.5 Zero Arity Functions Memoization

In order to achieve linear time performance the mechanism which wraps code into `delay` closures needs to be modified to memoize forced values. Because data cannot be modified within ADTs, they can be treated as a pure data structure - a generalization that allows memoization. Every time `force` is called, the delayed value is converted from a closure to its result and stored back in the underlying data structure. Subsequent calls to `force` return the memoized result. This modification allows the fibonacci sequence generation code to run in linear time.

4. Monads

Monads provide a powerful abstraction that allows convenient implementation of complex command flow behavior[9]. In this section I implement a monadic framework in Common Lisp and integrate lazy algebraic types `maybe` and `list` to operate within this framework.

4.1 The Skeleton

In Haskell, the class type `Monad` has the following definition[4, 3]:

```
class  Monad m  where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

This structure can be represented in Common Lisp as a series of generic function definitions:

```
(defgeneric unit (object monad-class))
(defgeneric bind (monad fn))
(defgeneric seq (monad1 monad2))
(defgeneric fail (message monad-class))
```

The functions look very similar to their Haskell equivalents. One difference, however, is that `unit` and `fail` require an extra argument - `monad-class`. The extra argument is necessary because Common Lisp lacks Haskell's type inference machinery - without it, it wouldn't be possible to determine which type to apply `unit` and `fail` to.

In addition, I provide default methods for `seq` and `fail`, similarly to Haskell:

```
(defmethod seq (monad1 monad2)
  (bind monad1 (constantly monad2)))

(defmethod fail (message monad-class)
  (error message))
```

Note that `seq` is defined in terms of `bind`. Together, these generic functions provide a skeleton that can be used to implement `Maybe` and `List` monads.

4.2 Implementing Commonly Used Monads

Haskell users are familiar with many commonly used monads: `IO`, `Maybe`, `List`, `Cont`, etc. In this section I will implement two of these constructs - `Maybe` and `List`.

4.2.1 Maybe Monad

The `maybe` type can now be extended based on the monadic framework developed in the previous section. The simplest functions to implement are `unit` and `fail` - the argument to `unit` is wrapped in a `just` constructor and `nothing` is returned on error:

```
(defmethod unit (object (monad-class maybe))
  (just object))

(defmethod fail (message (monad-class maybe))
  nothing)
```

The bulk of the work is done in `bind`. If an object can be matched against `just x`, `bind` passes `x` for further evaluation. Otherwise, `nothing` is returned:

```
(defmethod bind ((monad maybe) fn)
  (ematch monad
    ((just x) (funcall fn x))
    (nothing nothing)))
```

The `maybe` monad is now defined. Its operation can be tested by creating a standard safe division operation - return `nothing` if there's an attempt to divide by zero, otherwise return `(just value)`:

```
(defun safe/ (x y)
  (mmatch (x y)
          ((* 0) nothing)
          ((i j) (just (/ i j)))))
```

Chaining these computations in various ways will verify that the monadic code is operational:

```
(defun @concat (l1 l2)
  (mmatch (l1 l2)
          ((@nil ys) ys)
          (((@cons x xs) ys)
              (@cons x (@concat xs ys)))))

(defun @concat-lists (list)
  (match list
         (@nil @nil)
         ((@cons x xs)
             (@concat x (@concat-lists xs)))))
```

Figure 2. Additional list utilities used to implement the `list` monad.

```
> (bind (safe/ 10 5) (lambda (x) (safe/ x 2)))
=> (just 1)
> (bind (safe/ 10 0) (lambda (x) (safe/ x 2)))
=> nothing
```

From this small example it is evident that explicit chaining is not practical - it is overly verbose and difficult to manage. Syntactic sugar for chaining monads will be developed in section 4.3 in a form of `do`-notation.

4.2.2 List Monad

The implementation of a `list` monad is similar to the implementation of `maybe`. However, it allows far more interesting applications. In this section I'll implement a `list` monad in Common Lisp and use it to solve a non-deterministic equation.

First, it is necessary to extend utilities built for the `@list` type based on the ADT framework developed earlier. Figure 2 presents two functions - `concat`, and `concat-lists`. The former is equivalent to Haskell's `++` - it concatenates two lists. The latter is equivalent to Haskell's `concat`. It accepts a list of lists and concatenates their elements. Monad constructs `unit` and `fail` are easily implemented. For `unit` a value is wrapped in a list and for `fail` an empty list, `@nil`, is returned:

```
(defmethod unit (object (monad-class @list))
  (@list object))

(defmethod fail (message (monad-class @list))
  @nil)
```

Implementation of `bind` maps over the elements of the list and concatenates the results:

```
(defmethod bind ((monad @list) fn)
  (@concat-lists (@map fn monad)))
```

A simple monadic list computation can be obtained by evaluating the following expression:

```
> (seq (@list 1 2 3) (@list 4 5 6))
=> #A[4 5 6 4 5 6 4 5 6]
```

With this framework in place, the list monad can be used to find solutions to interesting problems. Figure 3 presents code that solves the following equation in monadic style:

$$x + y \geq 10 \,|\, x \in \{1, 2, 3, 4\};\, y \in \{5, 6, 7, 8\}$$

Evaluating the expression produces the following result:

```
#A[(2 . 8) (3 . 7) (3 . 8) (4 . 6) (4 . 7) (4 . 8)]
```

This is an exhaustive list of solutions to the equation presented above.

```
(bind (@list 1 2 3 4)
      (lambda (x)
        (bind (@list 5 6 7 8)
              (lambda (y)
                (if (< (+ x y) 10)
                    (fail "Sorry" @nil)
                    (unit (cons x y)
                          @nil))))))
```

Figure 3. Solving an equation with a list monad.

```
(defmacro with-monad (&body exprs)
  (let ((exp (car exprs)))
    (if (<= (length exprs) 1)
        exp
        (if (equalp (car exp) 'as)
            `(bind ,(caddr exp)
                   (lambda (,(cadr exp))
                     (with-monad ,@(cdr exprs))))
            `(seq ,exp
                  (with-monad ,@(cdr exprs)))))))
```

Figure 4. A macro that implements `do`-notation.

However, the code in figure 3 leaves a lot to be desired. Instead of taking advantage of the monadic abstraction to make the solution more clear, it obfuscates the solution with boilerplate. In order to apply the list monad to practical problems *cl-monad* introduces `do`-notation - syntactic sugar that makes binding monadic computations less tedious. In the next section I will discuss the implementation of this notation, and use it to rewrite the solution for above equation. Additionally, I will use the `do`-notation to generate a powerset.

4.3 Do-notation

In order to avoid verbosity demonstrated in figure 3, an equivalent of Haskell's `do`-notation[4] must be developed. Implementing such a notation in Common Lisp is difficult because of an abundance of special forms. A careful analysis needs to be performed of how the `do`-notation transformer should behave when it encounters any given form. At this time I've implemented a limited version in a form of `with-monad` macro (shown in figure 4), as such analysis is outside of scope of this paper. The macro `with-monad` behaves in a manner very similar to Haskell's operator `do`. It sequences monadic expressions using `seq`. Each expression is examined for a special symbol `as` that allows binding expressions to named variables. If such a symbol is found, `seq` is replaced with `bind` and a λ-function is generated with appropriate arguments.

The code in figure 3 can now be rewritten using `with-monad` macro:

```
> (with-monad
     (as x (@list 1 2 3 4))
     (as y (@list 5 6 7 8))
     (if (< (+ x y) 10)
         (fail "Sorry" @nil)
         (unit (cons x y) @nil)))
=> #A[(2 . 8) (3 . 7) (3 . 8) (4 . 6)
      (4 . 7) (4 . 8)]
```

This allows obtaining the same results with much cleaner code.

Now that `with-monad` macro is implemented, it can be used to solve one more problem with the list monad - generation of a powerset. An elegant solution to this problem requires introduction

```
(defun filterm (fn list monad-class)
  (mmatch (fn list)
          ((* @nil) (unit @nil @nil))
          ((p (@cons x xs))
              (with-monad
                (as flg (funcall p x))
                (as ys (filterm p xs monad-class))
                (unit (if flg (@cons x ys) ys)
                      monad-class)))))
```

Figure 5. An implementation of a Lisp alternative to `filterM`.

of one more utility function - `filterm`, an equivalent of Haskell's `filterM`. This function acts as a generalization of `filter` for monads. It is presented in figure 5. Note that `filterm` is a one to one equivalent to its Haskell alternative. The only difference is a third argument - `monad-class`, which must be introduced because Common Lisp lacks type inference machinery.

Once `filterm` is introduced it is trivial to apply it in order to generate a powerset. A Haskell solution for powerset generation is expressed with the following code[2]:

```
> filterM (const [True, False])
          [1, 2, 3]
=> [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

An equivalent *cl-monad* solution can be defined with very few modifications:

```
> (filterm (constantly (@list t nil))
           (@list 1 2 3)
           @nil)
=> #A[[1 2 3] [1 2] [1 3] [1] [2 3] [2] [3] []]
```

5. Future Work

For practical development, Haskell introduces strictness annotations that reduce thunking and allow fine grained control of evaluation semantics. Introducing such a system to *cl-monad* is an area that needs further work. Both ADT strictness declarations and eager pattern matching alternatives need to be introduced in order to solve common problems associated with lazy evaluation.

Additionally, further work is needed to extend the `with-monad` macro to behave correctly when it encounters many of Common Lisp's special forms. Simple type inference mechanisms may be integrated into `with-monad` that might allow automatic determination of the additional `monad-class` parameter. The parameter may be set as a special variable to avoid explicit user interaction.

Monad transformers is another area that requires additional work as combining distinct monadic types is outside the scope of this paper. Currently monadic transformers are not necessary in *cl-monad* because the type of the monad is passed to supporting functions explicitly. However, upon introduction of type inference technology within the `with-monad` macro, monad transformers will become a necessity.

A. Custom Pattern Matching

```
(defun define-algebraic-matcher
    (constructor-name arg-count)
  (if (> arg-count 0)
      `(define-constructor-matcher
         ,constructor-name
         ,arg-count
         (lambda (form)
           (m%when (typep form ',constructor-name)
             (apply #'values
                    (mapslots (lambda (slot-name)
                      (slot-value form slot-name))
                      form)))))
    `(define-symbol-matcher ,constructor-name
       `(lambda (form)
          (m%when
            (typep form
                   (type-of ,',constructor-name))
            (fare-matcher::m%success))))))
```


Acknowledgments

Thanks to all the good people on #lisp and #haskell IRC channels without whom this work wouldn't be possible.

References

[1] Paul Graham, *Ansi common lisp*, Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

[2] HaskellWiki, *Blow your mind*, http://www.haskell.org/haskellwiki/Blow_your_mind 2007.

[3] Paul Hudak, *The haskell school of expression: learning functional programming through multimedia*, Cambridge University Press, New York, NY, USA, 2000.

[4] Simon Peyton Jones (ed.), *Haskell 98 language and libraries: The revised report*, http://haskell.org/, September 2002.

[5] Peter Norvig, *Paradigms of artificial intelligence programming: Case studies in common lisp*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[6] Fare Rideau, *fare-matcher*, http://www.cliki.net/fare-matcher, 2005.

[7] P. Seibel, *Practical common lisp*, Apress, Berkeley, CA, USA, 2005.

[8] Shine, *Monads in lisp*, http://paste.lisp.org/display/44120, 2007.

[9] P. L. Wadler, *Comprehending monads*, Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice (New York, NY), ACM, 1990, pp. 61–78.