Lazy Scheme

Version 4.0.2

July 4, 2008

```
#lang lazy
```

Lazy Scheme is available as both a language level and a module that can be used to write lazy code. To write lazy code, simply use `lazy` as your module's language:

```
#lang lazy
... lazy code here...
```

Function applications are delayed, and promises are automatically forced. The language provides bindings that are equivalent to most of the `mzscheme` and `scheme/list` libraries. Primitives are strict in the expected places; struct constructors are lazy; `if`, `and`, `or` etc. are plain (lazy) functions. Strict functionality is provided as-is: `begin`, I/O, mutation, parameterization, etc. To have your code make sense, you should chain side effects in `begin`s, which will sequence things properly. (Note: This is similar to threading monads through your code—only use `begin` where order matters.)

Mizing lazy and strict code is simple: you just write the lazy code in the lazy language, and strict code as usual. The lazy language treats imported functions (those that were not defined in the lazy language) as strict, and on the strict side you only need to force (possibly recursively) through promises.

A few side-effect bindings are provided as-is. For example, `read` and `printf` do the obvious thing—but note that the language is a call-by-need, and you need to be aware when promises are forced. There are also bindings for `begin` (delays a computation that forces all subexpressions), `when`, `unless`, etc. There are, however, less reliable and might change (or be dropped) in the future.

There are a few additional bindings, the important ones are special forms that force strict behaviour—there are several of these that are useful in forcing different parts of a value in different ways, as described in §2 "Forcing Values".

1 Lazy Forms and Functions

`lambda`

Lazy variant of `lambda`.

`define`

Lazy variant of `define`.

`let`

Lazy variant of `let`.

`let*`

Lazy variant of `let*`.

`letrec`

Lazy variant of `letrec`.

`parameterize`

Lazy variant of `parameterize`.

`define-values`

Lazy variant of `define-values`.

`let-values`

Lazy variant of `let-values`.

`let*-values`

Lazy variant of `let*-values`.

`letrec-values`

Lazy variant of `letrec-values`.

`if`

Lazy variant of `if`.

`set!`

Lazy variant of `set!`.

`begin`

Lazy variant of `begin`.

`begin0`

Lazy variant of `begin0`.

`when`

Lazy variant of `when`.

`unless`

Lazy variant of `unless`.

`cond`

Lazy variant of `cond`.

`case`

Lazy variant of `case`.

`values : procedure?`

Lazy variant of values.

---

make-struct-type : procedure?

Lazy variant of make-struct-type.

---

cons : procedure?

Lazy variant of cons.

---

list : procedure?

Lazy variant of list.

---

list* : procedure?

Lazy variant of list*.

---

vector : procedure?

Lazy variant of vector.

---

box : procedure?

Lazy variant of box.

---

and : procedure?

Lazy variant of and.

---

or : procedure?

Lazy variant of or.

---

set-mcar! : procedure?

Lazy variant of set-mcar!.

`set-mcdr!` : `procedure?`

Lazy variant of `set-mcdr!`.

`vector-set!` : `procedure?`

Lazy variant of `vector-set!`.

`set-box!` : `procedure?`

Lazy variant of `set-box!`.

`error` : `procedure?`

Lazy variant of `error`.

`printf` : `procedure?`

Lazy variant of `printf`.

`fprintf` : `procedure?`

Lazy variant of `fprintf`.

`display` : `procedure?`

Lazy variant of `display`.

`write` : `procedure?`

Lazy variant of `write`.

`print` : `procedure?`

Lazy variant of `print`.

`eq?` : `procedure?`

Lazy variant of `eq?`.

`eqv?` : `procedure?`

Lazy variant of `eqv?`.

`equal?` : `procedure?`

Lazy variant of `equal?`.

`list?` : `procedure?`

Lazy variant of `list?`.

`length` : `procedure?`

Lazy variant of `length`.

`list-ref` : `procedure?`

Lazy variant of `list-ref`.

`list-tail` : `procedure?`

Lazy variant of `list-tail`.

`append` : `procedure?`

Lazy variant of `append`.

`map` : `procedure?`

Lazy variant of `map`.

`for-each` : `procedure?`

Lazy variant of `for-each`.

`andmap` : `procedure?`

Lazy variant of andmap.

---

ormap : procedure?

Lazy variant of ormap.

---

member : procedure?

Lazy variant of member.

---

memq : procedure?

Lazy variant of memq.

---

memv : procedure?

Lazy variant of memv.

---

assoc : procedure?

Lazy variant of assoc.

---

assq : procedure?

Lazy variant of assq.

---

assv : procedure?

Lazy variant of assv.

---

reverse : procedure?

Lazy variant of reverse.

---

caar : procedure?

Lazy variant of caar.

---

cadr : procedure?

Lazy variant of cadr.

---

cdar : procedure?

Lazy variant of cdar.

---

cddr : procedure?

Lazy variant of cddr.

---

caaar : procedure?

Lazy variant of caaar.

---

caadr : procedure?

Lazy variant of caadr.

---

cadar : procedure?

Lazy variant of cadar.

---

caddr : procedure?

Lazy variant of caddr.

---

cdaar : procedure?

Lazy variant of cdaar.

---

cdadr : procedure?

Lazy variant of cdadr.

---

cddar : procedure?

Lazy variant of cddar.

---

cdddr : procedure?

Lazy variant of cdddr.

---

caaaar : procedure?

Lazy variant of caaaar.

---

caaadr : procedure?

Lazy variant of caaadr.

---

caadar : procedure?

Lazy variant of caadar.

---

caaddr : procedure?

Lazy variant of caaddr.

---

cadaar : procedure?

Lazy variant of cadaar.

---

cadadr : procedure?

Lazy variant of cadadr.

---

caddar : procedure?

Lazy variant of caddar.

---

cadddr : procedure?

Lazy variant of cadddr.

---

cdaaar : procedure?

Lazy variant of cdaaar.

---

cdaadr : procedure?

Lazy variant of cdaadr.

---

cdadar : procedure?

Lazy variant of cdadar.

---

cdaddr : procedure?

Lazy variant of cdaddr.

---

cddaar : procedure?

Lazy variant of cddaar.

---

cddadr : procedure?

Lazy variant of cddadr.

---

cdddar : procedure?

Lazy variant of cdddar.

---

cddddr : procedure?

Lazy variant of cddddr.

---

first : procedure?

Lazy variant of first.

---

second : procedure?

Lazy variant of second.

`third` : `procedure?`

Lazy variant of `third`.

`fourth` : `procedure?`

Lazy variant of `fourth`.

`fifth` : `procedure?`

Lazy variant of `fifth`.

`sixth` : `procedure?`

Lazy variant of `sixth`.

`seventh` : `procedure?`

Lazy variant of `seventh`.

`eighth` : `procedure?`

Lazy variant of `eighth`.

`rest` : `procedure?`

Lazy variant of `rest`.

`cons?` : `procedure?`

Lazy variant of `cons?`.

`empty` : `procedure?`

Lazy variant of `empty`.

`empty?` : `procedure?`

Lazy variant of `empty?`.

`foldl` : `procedure?`

Lazy variant of `foldl`.

`foldr` : `procedure?`

Lazy variant of `foldr`.

`last-pair` : `procedure?`

Lazy variant of `last-pair`.

`remove` : `procedure?`

Lazy variant of `remove`.

`remq` : `procedure?`

Lazy variant of `remq`.

`remv` : `procedure?`

Lazy variant of `remv`.

`remove*` : `procedure?`

Lazy variant of `remove*`.

`remq*` : `procedure?`

Lazy variant of `remq*`.

`remv*` : `procedure?`

Lazy variant of `remv*`.

`memf` : `procedure?`

Lazy variant of memf.

---

assf : procedure?

Lazy variant of assf.

---

filter : procedure?

Lazy variant of filter.

---

sort : procedure?

Lazy variant of sort.

---

true : procedure?

Lazy variant of true.

---

false : procedure?

Lazy variant of false.

---

boolean=? : procedure?

Lazy variant of boolean=?.

---

symbol=? : procedure?

Lazy variant of symbol=?.

---

compose : procedure?

Lazy variant of compose.

---

build-list : procedure?

Lazy variant of build-list.

`take` : `procedure?`

Lazy variant of `take`.

`identity` : `procedure?`

Lazy identity function.

`cycle` : `procedure?`

Creates a lazy infinite list given a list of elements to repeat in order.

2 Forcing Values

```
(require lazy/force)
```

The bindings of `lazy/force` are re-provided by `lazy`.

```
(! expr) → any/c
  expr : any/c
```

Evaluates *expr* strictly. The result is always forced, over and over until it gets a non-promise value.

```
(!! expr) → any/c
  expr : any/c
```

Similar to `!`, but recursively forces a structure (e.g: lists).

```
(!!! expr) → any/c
  expr : any/c
```

Similar to `!!`, but also wraps procedures that if finds so their outputs are forced (so they are useful in a strict world).

```
(!list expr) → list?
  expr : (or/c promise? list?)
```

Forces the *expr* which is expected to be a list, and forces the `cdr`s recursively to expose a proper list structure.

```
(!!list expr) → list?
  expr : (or/c promise? list?)
```

Similar to `!list` but also forces (using `!`) the elements of the list.

2.1 Multiple values

To avoid dealing with multiple values, they are treated as a single tuple in the lazy language. This is implemented as a `multiple-values` struct, with a `values` slot.

```
(split-values x) → any
  x : multiple-values?
```

Used to split such a tuple to actual multiple values. (This may change in the future.)

```
(!values expr) → any
  expr : (or/c promise? multiple-values?)
```

Forces *expr* and uses `split-values` on the result.

```
(!!values expr) → any
  expr : (or/c promise? multiple-values?)
```

Similar to `!values`, but forces each of the values recursively.

3 Promises

```
(require lazy/promise)
```

The lazy/promise module implements lazy promises as implicitly used by the lazy language.

Note: this module implements a new kind of promises. MzScheme's promises are therefore treated as other values—which means that they are not forced by this module's force.

Generally speaking, if you use only delay, force, and promise?, you get the same functionality as in Scheme. See below for two (relatively minor) differences.

lazy implements a new kind of promise. When used with expressions, it behaves like delay. However, when lazy is used with an expression that already evaluates to a promise, it combines with it such that force will go through both promises. In other words, (lazy *expr*) is equivalent to (lazy (lazy *expr*)). The main feature of this implementation of promises is that lazy is safe-for-space (see SRFI-45 for details)—this is crucial for tail-recursion in Lazy Scheme.

To summarize, a sequence of lazys is forced with a single use of force, and each additional delay requires an additional force—for example, (lazy (delay (lazy (delay (lazy *expr*))))) requires three forces to evaluate *expr*.

Note: lazy cannot be used with an expression that evaluates to multiple values. delay, however, is fine with multiple values. (This is for efficiency in the lazy language, where multiple values are avoided.)

As mentioned above, using delay and force is as in Scheme, except for two differences. The first is a technicality—force is an identity for non-promise values. This makes it more convenient in implementing the lazy language, where there is no difference between a value and a promise.

The second difference is that circular (re-entrant) promises are not permitted (i.e., when a promise is being forced, trying to force it in the process will raise an error). For example, the following code (see srfi-45 for additional examples):

```
(let ([count 5])
  (define p
    (delay (if (<= count 0)
               count
               (begin (set! count (- count 1))
                      (force p)))))
  (force p))
```

returns 0 with Scheme's delay/force, but aborts with an error with this module's promises. This restriction leads to faster code (see a SRFI-45 discussion post for some additional de-

tails), while preventing diverging code (the only reasonable way to use circular promises is using mutation as above).

`(delay expr)`

Similar in functionality to Scheme's `delay`

`(lazy expr)`

Creates a "lazy" promise. See above for details.

`(force x)` → `any/c`
  `x` : `any/c`

Forces a promise that was generated by `delay` or `lazy`. Similar to Scheme's `force`, except that non-promise values are simply returned.

`(promise? x)` → `boolean?`
  `x` : `any/c`

A predicate for promise values.

4 MzScheme without Promises

```
(require lazy/mz-without-promises)
```

The `lazy/mz-without-promises` module simply provides all of `mzscheme` except for promise-related functionality: `delay`, `force`, and `promise?`. This is because `lazy/promise` defines and provides the same names. It is intended as a helper, but you can use it together with `lazy/promise` to get a `mzscheme`-like language where promises are implemented by `lazy/promise`.