

Morc

Mock Arc Programming Language as Scheme Extension

Version 0.1
2008-08-31

neil@neilvandyke.org

<http://www.neilvandyke.org/morc/>

Copyright © 2008 Neil Van Dyke. This program is Free Software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See <http://www.gnu.org/copyleft/lesser.html> for details. For other licenses and consulting, contact the author.

1 Introduction

Note: This is an incomplete implementation that I do not plan to pursue further. It is being released only for purposes of publicly archiving the work thus far, in case someone in the future might benefit from it. The current state of this is the result of two weekends of work, and then it was set aside due to other demands on my time. This work was done with PLT 360, and looks like minor tweaks would be required to get it to work with PLT 4. As I recall, I was last working on a namespace issue with PLT defmacro.

At the end of 2008-01, Paul Graham and Robert Morris made the initial release of the [Arc programming language](#). There was no reference documentation, and as I read through the tutorial text file, I noticed a striking resemblance of Arc to Scheme – Scheme, with lots of syntactic sugar, some Common Lisp-isms, some clever conveniences and shorthands that were inconsistent with Scheme. I strongly suspected that Scheme was used as a starting point for Arc. The similarities of Arc to Scheme inspired me to implement Arc as a set of Scheme macros and using a few extra features of PLT Scheme. It was in the course of implementing much of Arc that I noticed the significant differences. I decided to call my implementation Morc, as in “mock Arc,” as in an imitation Arc.

The Morc implementation is cleanroom, based only on the Arc tutorial, what little tidbits I’d noticed in public information, and my own knowledge of various programming languages and theory.

Because Morc worked by expanding Arc code to PLT Scheme code, the PLT compilers could be used, which, incidentally, I suspected would make Morc much faster than the Arc reference implementation. That is not a criticism of Arc, as the authors stated at the time that the implementation was to be more of an executable informal specification than efficient.

I was planning on having the API reference for Morc double as a commentary on how Arc concepts relate to those in Scheme and other languages. Two perceptions I noted at the time were “Arc seems to value terseness, whereas Scheme values purity,” and the even more inciting “You can do Arc in [PLT] Scheme, but not Scheme in Arc.”

File `test-morc.arc` is the beginning of a test suite for Arc, derived from the original Arc tutorial.

Here are some instructions I wrote while developing this under PLT 360, which might or might not be correct. “To use, install the Morc `.plt` file. (If you don’t know how to install a `.plt` file, see <http://download.plt-scheme.org/doc/dotplt.html>.) Start DrScheme. Select **Choose Language...** from the **File** menu, and choose **Morc**. Morc can also be invoked as `mred -Z -z -M morc`, such as from within **Quack**.

Please note that I’ve not touched any of the documentation, other than to add this Introduction for the release. “!!!” is my notation meaning that the documentation there needs work before release (since three exclamation marks together should never occur under any circumstance), and is usually followed by cryptic notes. Some of those notes allude to points I was intending to make in the annotated Morc reference documentation.

Good night, and good luck.

2 nil and Booleans

!!! “Some implementations provide variables `nil` and `t` whose values in the initial environment are `#f` and `#t` respectively.” [R3RS sec. 6.1]

`t` [Variable]
Mostly same as Scheme, except it’s spelled `t` instead of `#t`, and in Scheme it is not a variable. (Scheme’s choice has the advantage of not using a single-letter symbol, which conceivably might come in handy when single letter symbols are used to represent letters in symbolic programming.)

`nil` [Variable]
In Arc, this is both false and the null list value, whereas Scheme has distinguished the two for a long time. Morc secretly uses Scheme null list in its representation of pairs, but exposes the null list only as `nil` through Arc. Scheme’s `#f` is not a variable.

`no val` [Procedure]
Same as Scheme, except Arc boolean values are used instead of Scheme’s.

3 Reader and Quoting

The Arc reader is the same as R5RS Scheme’s. Morc uses the MzScheme reader, but with some features disabled.

`: +{ proc }` [Syntax]
!!! not-arc

4 Types

`type x` [Procedure]

!!! scheme has predicate procedures. you can ask "number?" or "integer?"

!!! though not shown in arc example, we return nil if we don't know type.

!!! why spell out "string" but not abbreviate "number" as "num"?

`isa x typ` [Procedure]

!!! achtung. arc tutorial defines num and int types, and says `isa` is

```
(def isa (x y) (is (type x) y))
```

which means that:

```
(isa 42 'int) ⇒ t
```

```
(isa 42 'num) ⇒ nil
```

!!! scheme integers answer both to `integer?` and `number?`

```
(integer? 42) ⇒ #t
```

```
(number? 42) ⇒ #t
```

`coerce x typ ?{ extra }` [Procedure]

5 Numbers

`even number` [Procedure]

`odd number` [Procedure]

Same as Scheme `even?` and `odd?`, respectively, except they return Arc boolean values.

`< num1 +{ num }` [Procedure]

`> num1 +{ num }` [Procedure]

`<= num1 +{ num }` [Procedure]

`>= num1 +{ num }` [Procedure]

Same as in Scheme, except they return Arc boolean values.

!!! While not required for Arc tutorial, Morc extends these to also work on strings.

`expt x y` [Procedure]

`sqrt x` [Procedure]

Same as in Scheme.

`++ var` [Procedure]

`-- var` [Procedure]

6 Strings

`string *{ arg }` [Procedure]

!!! we convert lists and pairs together

7 Lists

`cons car cdr` [Procedure]
`car pair` [Procedure]
`cdr pair` [Procedure]
`list *{ arg }` [Procedure]

Same as in Scheme, except that the Arc `nil` object is used instead of the Scheme `()` null list.

`cadr pair` [Procedure]
!!! why have this when arc has `car:cdr` composition? the old lisp `ca*d*r` procedures could be seen a kludge around the lack of syntactic sugar.

$(\text{cadr } x) \equiv (\text{car}:\text{cdr } x)$

`nthcdr n lst` [Procedure]
`firstn n lst` [Procedure]

`tuples lst ?{ size }` [Procedure]
!!! Scheme has none. Morc's implementation is in theory twice as efficient as the one in the Arc0 tutorial, and is implemented using `named-let` and multiple-value returns. (example of why powerful fundamentals important to good algorithms)

!!! first need fundamentals, rather than composing out of fancy primitives that might specify the functional behavior but aren't very good algorithmically. example is "tuples"

`push val list-var` [Procedure]
`pop list-var` [Procedure]

8 Association Lists

`alref alist key` [Procedure]
!!! what about not-found value or not-found thunk? in morc, we've currently defined it to yield `nil` if key not found. in scheme, you'd use `assq`, `assv`, and `assoc`.

9 Hash Tables

`table` [Procedure]
`listtab lst` [Procedure]
`obj *{ key val }` [Syntax]
`keys ht` [Procedure]
`vals ht` [Procedure]

!!! `plt-scheme` has `hash-table-map`, which is better

`mactable proc ht` [Procedure]
!!! Why call it “map” if it doesn’t function analogously to `map`? And why not return the result of mapping via `proc`? “There is a function called `mactable` for hash tables that is like `map` for lists, except that it returns the original table instead of a new one.”

10 Equality and Identity

`is obj1 obj2` [Procedure]

`iso obj1 obj2` [Procedure]
!!!

`in key *{ item }` [Procedure]
!!!

11 Closures

`fn (*{ arg }) *{ body }` [Syntax]
Same as Scheme `lambda`, without the apparent ability [!!!] to collect the rest of a list of values into one argument.
!!! optional args is like `opt-lambda`, just with extraneous "o"

12 Application

!!! scheme list sexy syntax is either a special form or a procedure application. arc permits...

`apply func args` [Procedure]
!!! does special dispatch. unclear whether Arc requires this.

13 Binding and Assignment

`def !!!` [Syntax]

`let !!!` [Syntax]
!!! we’ll make this a `letrec` ... (does that help self-recursive `lambda`?)

`with !!!` [Syntax]
!!! we’ll make this a `letrec` ..., but maybe it should be a `let*`

`= !!!` [Syntax]

14 Conditionals

<code>do !!!</code>	[Syntax]
<code>and !!!</code>	[Syntax]
<code>or !!!</code>	[Syntax]
<code>if !!!</code>	[Syntax]
<code>when !!!</code>	[Syntax]
!!! mention i defined when and unless for mzscheme, and now no longer use them.	
<code>case !!!</code>	[Syntax]

15 Iteration

<code>each !!!</code>	[Syntax]
<code>for !!!</code>	[Syntax]
<code>while <i>expr</i> *{ <i>body</i> }</code>	[Syntax]
<code>repeat <i>num</i> +{ <i>body</i> }</code>	[Syntax]

16 List Processing

<code>map <i>func</i> +{ <i>lst</i> }</code>	[Procedure]
<code>keep <i>pred list-or-string</i></code>	[Procedure]
<code>rem <i>pred list-or-string</i></code>	[Procedure]
!!! note that arc doesn't say whether the equality comparison is "is" or "iso".	
!!! "keep" with equality is a little odd	
<code>all !!!</code>	[Procedure]
<code>some !!!</code>	[Procedure]
!!!	
<code>pos !!!</code>	[Procedure]
!!! unclear when would use this, since you don't have direct access to list elements by index. makes more sense for string functions, especially when there is efficient direct access to string elements (not necessarily the case with non-ASCII strings).	
<code>true<i>s</i> !!!</code>	[Procedure]

17 Input and Output

<code>pr !!!</code>	[Procedure]
<code>prn !!!</code>	[Procedure]
<code>tostring +{ <i>body</i> }</code>	[Syntax]

18 Misc.

`len` *x* [Procedure]

`sort` *less-than lst* [Procedure]

!!! use `sort` function included with `plt-scheme`, albeit with arguments reversed.

`compare` *less-than convert* [Procedure]

19 Macros

`mac` *name args* +{ *body* } [Syntax]

`uniq` [Procedure]

Morc implements this by simply calling MzScheme's (non-Scheme) Lisp `gensym`.

`w/uniq` *VARs* [Syntax]

20 Non-Arc Utilities

`.expand` *expr* [Procedure]

`.expand1` *expr* [Procedure]

`.null` [Variable]

21 Kludge

22 Namespace

23 Tests

Some tests are in the file `test-morc.arc`, which is included in the Morc installation package.

History

Version 0.1 — 2008-08-31

First release, and only intended release.