Lisp in Summer Projects Submission

Submission Date	2013-10-25 09:19:58
Full Name	John Boyle
Country	USA
Project Name	emiya
Type of software	command-line/terminal app
General category	lisp compiler/interpreter
LISP dialect	Arc
GitHub URL	https://github.com/waterhouse/emiya
Did you start this project?	No, I'm modifying or extending an existing project.
Which file or directory contains the majority of your work?	dyn-cont7.arc
Briefly describe your modifications	Actually I wrote everything except the *.arc files directly in arc3.1 (Arc was created by Paul Graham and Robert Morris), and modified some of the latter. But the point of this project is dyn-cont7. I've written a series of Arc interpreters in Arc; dyn-cont is definitely not the first; but the code was all written after the start of the competition--except for "de-macro" and "fn-optional".
Project Description	I want to describe my project in this form.
Purpose	My project has two parts: First, an interpreter for a form of Arc in which macros and special forms are all first-class objects. It provides continuations, parameters, and exceptions. Second, a user program running in the interpreter, which makes some interesting use of the first-class special forms.
Function	It defines the interpreter's primitives in the parent Arc. Then it loads a "standard library", a sequence of interpreted-Arc expressions, that defines increasingly sophisticated utilities

in terms of the primitives, and twice re-"compiles" (i.e. macroexpands) everything.

Motivation

I quote: "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."

Permitting first-class usage of macros and special forms seemed like the "right thing" to me, yet no major implementations did it. I wanted to demonstrate that it could be done, and to encourage others to adopt and extend my ideas. (Also, if others are impressed, that's nice.)

Audience

Well, firstly, myself. Second, anyone who is interested in language design and implementation. Third, anyone who wants a model of an Arc interpreter to implement in native code (probably mainly myself).

Methodology

At first glance, we have a classic Lisp interpreter in dyn-cont7.arc. The name comes from its support for dynamic scope (with Scheme-style parameters) and continuations.

The interpreter uses the cons cells, strings, characters, numbers, symbols, and a few primitive procedures of the parent Arc. Closures are represented as `(clos ,env ,body), macros as `(macro ,closure). Special forms are represented as, e.g., '(special-value . if). Continuations are single-argument parent-Arc procedures; parameters are `(dyn . ,value). Environments are represented as assoc-lists of (var val). Dynamic environments are assoc-lists of (dyn val). The global environment is a parent-Arc hash table.

The code is written in continuation-passing style (except for short subroutines). Most of the core evaluation functions accept a dynamic environment, a lexical environment, and a continuation. Thus, it provides continuations without using parent-Arc's call-cc, and parameters without parent-Arc (or rather parent-Racket)'s make-parameter; also, the current exception handler is a parameter. This explicitness is intended to provide easier translation to a low-level language.

The interesting feature of the interpreter is that it evaluates *all* compound expressions (...) by evaluating the --the car--and deciding what to do from there. This might seem obvious, but then you realize that it means there cannot be anything special about the symbol 'if, or 'quote, etc., nor about the names of macros; they are merely variables which happen to be bound to something special. Furthermore, they are necessarily equivalent to, e.g., (car xs), if (eval '(car xs)) produces the same result as (eval 'if).

First-class macros make "eval" simpler. The author is convinced they are fully on the right side of "removing weaknesses and restrictions" vs. "piling feature on top of feature".

--

So much for the interpreter. Now for the user program it runs. It begins, like arc.arc, by defining "def" and "mac" as soon as possible, along with other utilities.

The first real surprise probably comes when quasiquote is defined by the user as a macro. The author is pleased with his definition; he considers it nice, and believes its output to be optimal or not far off. (In arc3.1/interpreters-etc/, see quasiquote-macro5.arc for the normal source code, and the quasiquote-macro-primitive.arc series to see it reworked into the interpreter.)

Also, later on, fn (the equivalent of lambda) is redefined by the user as a macro (which expands to a call to "underlying-fn", bound to the old value of "fn"), to give it Arc's optional arguments. This would be impossible in most implementations--redefining 'lambda alone is often outright forbidden; getting a new symbol to behave like the old 'lambda can probably be done by importing 'lambda under a new name, but getting old functions to use the new macro defs may be outside its compilation model, and the source may be inaccessible. It all works fine in the interpreter; it just uses the current definition of each symbol it encounters.

But this comes at a price. Its functions always have fresh macroexpansions because it creates them fresh every time. Now, quasiquote and its subroutines are written with a lot of macros, in good Arc style. As a result, noted in the comments, it takes about a fifth of a second and generates megabytes of garbage to evaluate the expression:

```
(let x 1 `(+ ,x 2))
=> (+ 1 2)
```

This is quite intolerable. Fortunately, there are solutions.

The basic idea is to preexpand macros. A conventional implementation will expand macros in function bodies as soon as it sees them [or conceivably JIT when they are called]. However, it generally will also not recompile functions when macros are redefined at the REPL, which is something the author liked about the interpreter.

A proper compilation/recompilation system should probably do something like invalidating old compilation results when something gets redefined, then JIT-recompiling them once when necessary. The author felt this system was beyond his current reach, especially if he had to write it without using quasiquote; however, he did not want to bake an inferior solution into the interpreter. Therefore:

Conventional macroexpansion is implemented in the user program. The function "de-macro" does the heavy work. "def" and "mac" are redefined so that they first de-macroize their inputs before creating a closure (wrapped in a macro in the latter case). Then all existing functions and macros get de-macroized, with their source code saved elsewhere. To facilitate this, the interpreter provides accessors for the code and env fields of closures, and the closure field of macros, and constructors for both. (One could use this system to bootstrap a more proper one, replacing "def" and "mac" again.)

Well, after the definitions of quasiquote et al are de-macroized, (let x 1 `(+ ,x 2)) takes 1.2 msec and 16KB of garbage to evaluate. Thus, one can support a lavish programming style on a tight budget.

Conclusion

So, what have we learned? We've learned that giving macros and special forms first-class status yields a flexible (and arguably elegant) interpreter design that allows several policy and implementation choices to be made by the user program:

First, compilation and recompilation semantics. We start with no compilation whatsoever; we use that to bootstrap into having de-macroization, which is the minimum necessary for tolerable performance [it sped up a simple test case by a factor of ~175]. This could (should) be used in turn to bootstrap into something more sophisticated in the future, such as functions getting recompiled when a macro is redefined (we could redefine "assign" to check what it's (re)defining).

Second, argument lists. We started with an "fn" that supported only regular and rest args, and extended it to support optional arguments; this was made possible by "no symbols are sacred". It could be further extended to make keyword args.

Third, intra-symbol syntax is defined by the user program, as a byproduct of de-macroization. (The author's symbol syntax is slightly different from Arc 3.1's.) This feature of Arc is relatively new and undeveloped, and for the moment experimentation should probably be encouraged.

To increase performance further, calls to basic n-ary functions like + and < could be turned into their two-argument versions, as in Arc 3.1's compiler; this could form part of more general procedural inlining. Furthermore, native code compilation itself is putting together strings of bytes and putting them in an executable region of memory; this should also be a user program. (A fully fledged version of this implementation should have a tiny core, used to bootstrap, and a large user program that does everything else.)

Last, this thing should really be running in native code (not in Arc 3.1), and should use real-time garbage collection. See the "vaporware" directory for some indication of my plans.

Build Instructions

Install Racket from racket-lang.org and, if necessary, modify your $PATH so that the executable "racket" is accessible under that name. "rlwrap" is an optional luxury.

Test Instructions

In a terminal, change to the emiya directory, and run arc.sh. If it can't find "racket", it will complain; also it will suggest you install "rlwrap" for an improved REPL experience.

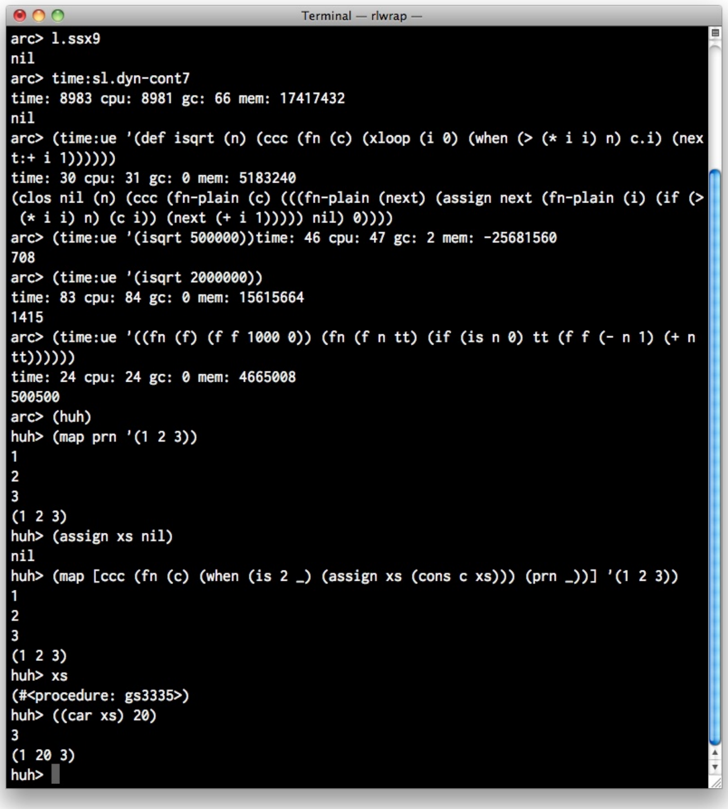
You will see some warnings mentioning "nasm" before the Arc prompt; it is trying to run "nasm" on some assembly files and load them. Likely those files and perhaps "nasm" don't exist on your computer. Don't worry about that, as long as you see "arc> " at the bottom. Enter (+ 1 2) to see that it works.

Execution Instructions

At the arc> prompt in the emiya directory, run
arc> l.ssx9
which loads the author's ssyntax, and then

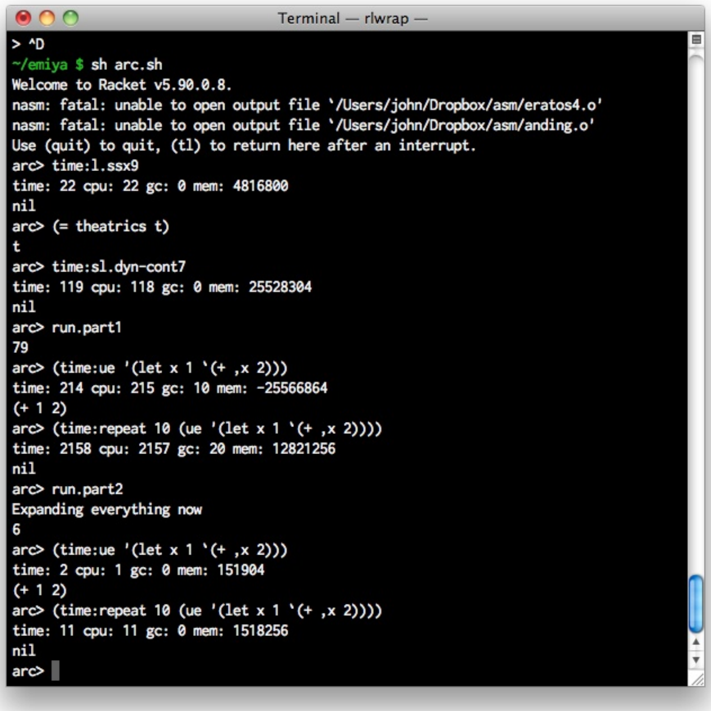
	arc> sl.dyn-cont7 which loads the interpreter and runs its user program. The latter may take ~10 seconds. (ue) evaluates "expr" in the interpreter. (huh) goes into a REPL with the interpreter. To benchmark the interpreter, I recommend (time:ue). Arc's "do" is like "begin" or "progn". If you want to see what happens before and after de-macroization, run this sequence of commands (see the "effects-of-macroexpansion" screenshot): l.ssx9 (= theatrics t) sl.dyn-cont7 run.part1 (time:ue '(let x 1 `(+ ,x 2)))  part1 contains everything just before de-macroization. Feel free to try other expressions; big quasiquote expressions are amazing. Then try: run.part2 (time:ue '(let x 1 `(+ ,x 2))) Observe the giant difference. To count to 1000 (and compute a triangular number) in Arc, I recommend: (xloop (i 1000 n 0) (if (is i 0) n (next (- i 1) (+ i n))))
Describe any bugs or caveats	You will see some warnings before the Arc prompt; it is trying to run "nasm" on some assembly files and load them. Likely those files and perhaps "nasm" don't exist on your computer. Don't worry about the warnings. In the interpreter, some errors will send you into the "huh> " prompt, others back into the "arc> " prompt. Such error handling was low priority to the author. Control-C will send you into a Racket prompt, and then, as Arc tells you, (tl) will get you back to "arc> ". The REPL doesn't expand symbol syntax at the moment, although de-macroizing an expression (explicitly, or implicitly through "def") does it.

Screen shots



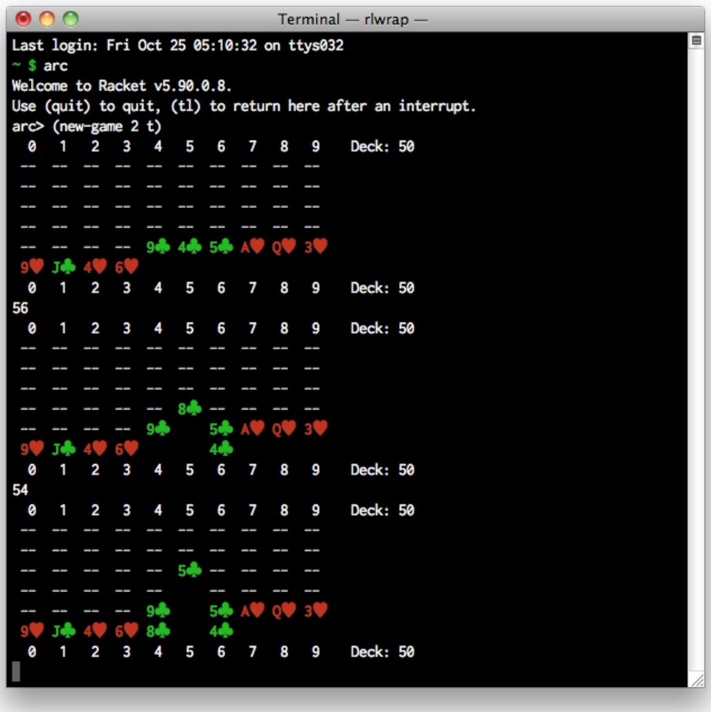
```
Terminal — rlwrap —
arc> l.ssx9
nil
arc> time:sl.dyn-cont7
time: 8983 cpu: 8981 gc: 66 mem: 17417432
nil
arc> (time:ue '(def isqrt (n) (ccc (fn (c) (xloop (i 0) (when (> (* i i) n) c.i) (nex
t:+ i 1))))))
time: 30 cpu: 31 gc: 0 mem: 5183240
(clos nil (n) (ccc (fn-plain (c) (((fn-plain (next) (assign next (fn-plain (i) (if (>
 (* i i) n) (c i)) (next (+ i 1))))) nil) 0))))
arc> (time:ue '(isqrt 500000))time: 46 cpu: 47 gc: 2 mem: -25681560
708
arc> (time:ue '(isqrt 2000000))
time: 83 cpu: 84 gc: 0 mem: 15615664
1415
arc> (time:ue '((fn (f) (f f 1000 0)) (fn (f n tt) (if (is n 0) tt (f f (- n 1) (+ n
tt))))))
time: 24 cpu: 24 gc: 0 mem: 4665008
500500
arc> (huh)
huh> (map prn '(1 2 3))
1
2
3
(1 2 3)
huh> (assign xs nil)
nil
huh> (map [ccc (fn (c) (when (is 2 _) (assign xs (cons c xs))) (prn _))] '(1 2 3))
1
2
3
(1 2 3)
huh> xs
(#<procedure: gs3335>)
huh> ((car xs) 20)
3
(1 20 3)
huh>
```

continuations-work-ok.png



```
Terminal — rlwrap —
> ^D
~/emiya $ sh arc.sh
Welcome to Racket v5.90.0.8.
nasm: fatal: unable to open output file '/Users/john/Dropbox/asm/eratos4.o'
nasm: fatal: unable to open output file '/Users/john/Dropbox/asm/anding.o'
Use (quit) to quit, (tl) to return here after an interrupt.
arc> time:l.ssx9
time: 22 cpu: 22 gc: 0 mem: 4816800
nil
arc> (= theatrics t)
t
arc> time:sl.dyn-cont7
time: 119 cpu: 118 gc: 0 mem: 25528304
nil
arc> run.part1
79
arc> (time:ue '(let x 1 `(+ ,x 2)))
time: 214 cpu: 215 gc: 10 mem: -25566864
(+ 1 2)
arc> (time:repeat 10 (ue '(let x 1 `(+ ,x 2))))
time: 2158 cpu: 2157 gc: 20 mem: 12821256
nil
arc> run.part2
Expanding everything now
6
arc> (time:ue '(let x 1 `(+ ,x 2)))
time: 2 cpu: 1 gc: 0 mem: 151904
(+ 1 2)
arc> (time:repeat 10 (ue '(let x 1 `(+ ,x 2))))
time: 11 cpu: 11 gc: 0 mem: 1518256
nil
arc>
```

effects of macroexpansion.png



```
Terminal — rlwrap —
Last login: Fri Oct 25 05:10:32 on ttys032
~ $ arc
Welcome to Racket v5.90.0.8.
Use (quit) to quit, (tl) to return here after an interrupt.
arc> (new-game 2 t)
  0   1   2   3   4   5   6   7   8   9    Deck: 50
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  9♣  4♣  5♣  A♥  Q♥  3♥
 9♥  J♣  4♥  6♥
  0   1   2   3   4   5   6   7   8   9    Deck: 50
56
  0   1   2   3   4   5   6   7   8   9    Deck: 50
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  8♣  --  --  --  --
 --  --  --  --  9♣      5♣  A♥  Q♥  3♥
 9♥  J♣  4♥  6♥          4♣
  0   1   2   3   4   5   6   7   8   9    Deck: 50
54
  0   1   2   3   4   5   6   7   8   9    Deck: 50
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  --  --  --  --  --
 --  --  --  --  --  5♣  --  --  --  --
 --  --  --  --  --      --  --  --  --
 --  --  --  --  9♣      5♣  A♥  Q♥  3♥
 9♥  J♣  4♥  6♥  8♣      4♣
  0   1   2   3   4   5   6   7   8   9    Deck: 50
```

just for fun.png

things-work-ok.png

Official	I have read rules and have abided by them. I am 18 years of age or older. I am not living in Brazil, Quebec, Saudi Arabia, Cuba, Iran, Myanmar (Burma), North Korea, Sudan, or Syria.