First-class (run-time) macros and meta-circular evaluation

First-class macros are macros that can be bound to variables, passed as arguments and returned from functions. First-class macros expand and evaluate syntax at *run*-time. Meta-circular evaluators support a concise implementation of first-class macros. In fact, first-class macros are *easier* to implement than traditional compile-time macros. A **meta-circular evaluator** is an interpreter which (1) can evaluate itself and (2) implements each language construct in terms of itself. This article and the attached implementation explain how to implement first-class macros in a meta-circular evaluator.

You might also be interested in:

- Paul Graham's posts on Arc, the first place I saw first-class macros.
- John McCarthy's original meta-circular evaluator for Lisp
- Chapter 4 of the MIT textbook Structure and Interpretation of Computer Programs (SICP), which popularized meta-circular evaluation.

Macros

With classical Lisp macros, you could define `and` in terms of `if`:

```
(define-macro (and a b)
 `(if ,a ,b #f))
```

With first-class (a.k.a. run-time) macros, you would write `and` as:

```
(define and (macro (lambda (a b)
                    `(if ,a ,b #f))))
```

One advantage of the first-class form is that the programmer can pass first-class macros as arguments to procedures; passing a compile-time macro to a procedure causes a compile-time error.

Because of their first-class nature, first-class macros make it easy to add or simulate any degree of laziness.

Of course, run-time macros incur a run-time overhead. Fortunately, my research in static analysis leads me to believe that we will be able to erase that overhead for the common case--where first-class macros are used to do the job of compile-time macros.

The structure of interpretation

SICP teaches (and preaches) the eval/apply model of interpretation. In this model, interpretation toggles between two functions, `eval` and `apply`:

- `eval : exp × env → value`
- `apply : value × value`* `→ value`

The type `exp` contains expressions, such as numerals, variable references, applications and lambda terms. The type `env` maps variables to values:

```
env = variable → value
```

The type value contains run-time values, such as numbers, lists and closures.

The `eval` function evaluates an expression in the context of an environment for its free variables. The `apply` function applies a procedure to a list of arguments.

The foundational principle of Lisp that makes macros so elegant is the unification of code and data through S-Expressions, so that:

- `eval : s-exp × env → s-exp`
- `apply : s-exp × s-exp`* `→ s-exp`

Clearly, if this approach is to work, then the definition of S-Expressions must also include semantic values like closures; that is, there will be S-Expressions with no literal form.

A typical `eval` procedure

A meta-circular evaluator with first-class macros differs mostly in the structure of its `eval` procedure. A typical `eval` procedure for a meta-circular evaluator looks something like:

```
(define (eval exp env)
  (cond
    ((symbol? exp)    (env-lookup env exp))
    ((number? exp)    exp)
    ((boolean? exp)   exp)
    ((string? exp)    exp)
    ((quote? exp)     (eval-quote exp env))

    ((if? exp)        (eval-if exp env))
    ((cond? exp)      (eval-cond exp env))
    ((or? exp)        (eval-or exp env))
    ((and? exp)       (eval-and exp env))
    ((lambda? exp)    (eval-lambda exp env))
    ((let? exp)       (eval (let->app exp) env))
    ((let*? exp)      (eval (let*->let exp) env))
    ((letrec? exp)    (eval (letrec->lets+sets exp) env))
    ((begin? exp)     (eval-begin exp env))
    ((set!? exp)      (eval-set! exp env))

    ((app? exp)       (apply (eval (app->fun exp) env)
                             (map (lambda (arg) (eval arg env))
                                  (app->args exp))))))
```

A first-class macro `eval` procedure

The `eval` procedure in an implementation with first-class macros is shorter:

```
(define (eval exp env)
  (cond
    ((symbol? exp)    (env-lookup env exp))
    ((number? exp)    exp)
    ((boolean? exp)   exp)
    ((string? exp)    exp)

    ; 3D-syntax is invoked to produce a captured value:
    ((procedure? exp) (exp))

    ((app? exp)       (perform-apply (eval (app->fun exp) env)
                                     exp env))))
```

With first-class macros, the core syntactic forms are not privileged; rather, they are defined in the initial environment:

```
(define initial-environment-amap
  (list (list 'apply      apply)
        (list '+          +)
        (list 'not        not)
        (list 'display    display)
        (list 'newline    newline)
        (list 'cons       cons)
        (list 'car        car)
        (list 'cdr        cdr)
        (list 'cadr       cadr)
        (list 'caadr      caadr)
        (list 'cadar      cadar)
        (list 'cddr       cddr)
        (list 'cdddr      cdddr)
        (list 'list       list)
        (list 'null?      null?)
        (list 'pair?      pair?)
        (list 'list?      list?)
        (list 'number?    number?)
        (list 'string?    string?)
        (list 'symbol?    symbol?)
        (list 'procedure? procedure?)
        (list 'eq?        eq?)
        (list '=          =)
        (list 'gensym     gensym)
        (list 'void       void)

        (list 'quote    (list 'syntax-primitive eval-quote))
        (list 'if       (list 'syntax-primitive eval-if))
        (list 'cond     (list 'syntax-primitive eval-cond))
        (list 'and      (list 'syntax-primitive eval-and))
        (list 'or       (list 'syntax-primitive eval-or))
        (list 'let      (list 'syntax-primitive eval-let))
        (list 'let*     (list 'syntax-primitive eval-let*))
        (list 'letrec   (list 'syntax-primitive eval-letrec))
        (list 'begin    (list 'syntax-primitive eval-begin))
        (list 'set!     (list 'syntax-primitive eval-set!))
        (list 'lambda   (list 'syntax-primitive eval-lambda))
        (list 'macro    (list 'syntax-primitive eval-macro))))
```

Interpretation-level primitives are actually bound directly as their meta-level primitives.

The `perform-apply` procedure must check whether the operator to be applied is a syntactic primitive, a macro or a procedure:

```
(define (perform-apply fun app-exp env)
  (let ((args (app->args app-exp)))
    (cond
      ((macro? fun)       (eval (apply (macro->proc fun) args) env))
      ((syntax-prim? fun) ((syntax-primitive->eval fun) app-exp env))
      (else                 (let ((arg-values (eval* args env)))
                              (apply fun arg-values))))))
```

3D-syntax and hygiene

Some constructs, such as `let` and `letrec`, are desugared into other constructs, such as `lambda` and `set!`; for example:

```
 (let ((var exp) ...) body)
```

becomes:

```
 ((lambda (var ...) body) exp ...)
```

This sort of expansion can cause a problem if we use a `let` construct in a context where `lambda` has been redefined. For example, we might define a function to compute the energy of a photon:

```
(define (energy lambda)
 (let ((c speed-of-light)
       (h plancks-constant))
  (/ (* c h) lambda)))
```

When the `let` form expands into a `lambda`-application this code, the symbol `lambda` is no longer bound to the syntactic primitive for `lambda`; rather, it is bound to some numeral representing wavelength. When the evaluator tries to evaluate this code, it will throw a particularly cryptic error about trying to apply a non-function value. This kind of capture is one of the two kinds of "hygiene" violations that Lisp systems worry about, and it is the only one that cannot be solved with `gensym`.

The provided implementation solves this hygiene problem through 3D syntax. An expression is **3D** if a programmer cannot write it down. In other words, it is an expression that *must* have come from a special syntactic expansion. In Lisp, raw procedures are 3D, because there is no way to write down a literal that the `read` procedure will pull in as a procedure.

If you examine the `eval` procedure for the first-class macros implementation, you will find a case not present in the ordinary evaluator: `procedure?`. When the evaluator hits a procedure, it assumse it takes no arguments and then evaluates it, directly returning whatever that procedure returns.

This behavior provides a way to pass protected values out of first-class macros, since they will be evaluated in whatever scope there was when the closure was born. Consequently, a `let` from in my implementation (effectively) expands into:

```
((,(lambda () 3d-lambda) ,@var ,@body) ,@exp)
```

where `3d-lambda` is bound to the syntactic primitive for `lambda`.

Code

[meta-circ.scm]

```
 meta-circular evaluator with first-class macros.

uthor: Matthew Might
ite:   http://matt.might.net/
       http://www.ucombinator.org/

his evaluator runs in R5RS Scheme, or itself.



tilities.

nsym-count : integer
ine gensym-count 0)

nsym : symbol -> symbol
ine gensym (lambda params
             (if (null? params)
                 (begin
                   (set! gensym-count (+ gensym-count 1))
                   (string->symbol (string-append
                                    "$"
                                    (number->string gensym-count))))
                 (begin
                   (set! gensym-count (+ gensym-count 1))
                   (string->symbol (string-append
```

[meta-circ-tests.scm]

```
(load "meta-circ.scm")


;; Testing and debugging.

(define-syntax check-expect
  (syntax-rules ()
    ((_ check expect)
       (let ((checked check)
             (expected expect))
```

```
         (if (not (equal? checked expect))
             (begin
               (display "expression: ")
               (write (quote check))
               (newline)
               (display "received:   ")
               (write checked)
               (newline)
               (display "expected:   ")
               (write expected)
               (newline))
             (if #f #t))))))


(define-syntax check-expect-eval
  (syntax-rules ()
    ((_ check-exp expected)
```

Latest: HOWTO: Get tenure
Next: Architectures for interpreters
Prev: Programming with continuations by example
Rand: The shape of your problem