Macros and Syntax Transformations

CSE 410/510 ETH: Interactive Programming Environments

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo

Homoiconicity

Lisp code looks like lisp data.

`(read)` essentially returns an abstract syntax tree!

We call this homoiconicity.

This encourages `(eval (transform (read)))`.

A Trivial Transform

```
(defconstant tau 6.283185307)

(defun replace-pi (&rest body)
  (labels ((do-replace (l)
             (cond
               ((and (atom l) (eql l 'pi))
                (list '/ 'tau 2.0))
               ((atom l) l)
               (t (mapcar #'do-replace l)))))
    (cons 'progn (mapcar #'do-replace body))))

(eval (replace-pi '(* pi 1 1)))
```

Expansion Time

This construction is expanded and evaluated together.

Frequently we wish to:

- expand at “compile time”
- evaluate at “run time”

(These concepts are a little fuzzy in Lisp!)

defmacro

Common Lisp provides `defmacro`:

```
(defmacro my-when (pred &body body)
  `(if ,pred
       (progn
         ,@body)))

(macroexpand-1
  '(my-when (not (null l))
     (process (car l))
     (recurse (cdr l))))

(IF (NOT (NULL L))
    (PROGN (PROCESS (CAR L)) (RECURSE (CDR L))))
```

Surprise!

It is important that macros minimize surprise.

This means things like:

- Evaluating things in an order that preserves side effects
- Not (unexpectedly) evaluating things more than once
- Not introducing name clashes

This requires tools and careful design.

Multiple Evaluation

Consider this (modified) example from CLTL2 [1]:

```
(defmacro arithmetic-if (test neg-form zero-form
    pos-form)
  `(cond
     ((< ,test 0) ,neg-form)
     ((= ,test 0) ,zero-form)
     (t           ,pos-form)))
```

Note that `test` will be evaluated twice!

Using let

This can be avoided using `let`:

```
(defmacro arithmetic-if (test neg-form zero-form
    pos-form)
  `(let ((result ,test))
     (cond
       ((< result 0) ,neg-form)
       ((= result 0) ,zero-form)
       (t            ,pos-form))))
```

Name Conflicts

But now:

```
(let ((result (calculation)))
  (arithmetic-if result (* result -1) 0 result))
```

Name Conflicts

But now:

```
(let ((result (calculation)))
  (arithmetic-if result (* result -1) 0 result))

(LET ((RESULT (CALCULATION)))
  (LET ((RESULT RESULT))
    (COND ((< RESULT 0) (* RESULT -1)) ((= RESULT 0)
       0) (T RESULT))))
```

Which RESULT?

gensym

(gensym) produces a globally unique variable name.

```
(defmacro arithmetic-if (test neg-form zero-form
    pos-form)
  (let ((result (gensym)))
    `(let ((,result ,test))
       (cond
         ((< ,result 0) ,neg-form)
         ((= ,result 0) ,zero-form)
         (t             ,pos-form)))))
```

The Expansion

```
(LET ((RESULT (CALCULATION)))
  (LET ((#:G245 RESULT))
    (COND ((< #:G245 0) (* RESULT -1))
          ((= #:G245 0) 0)
          (T RESULT))))
```

Hygienic Macros

Hygienic macros [2] solve name conflicts in macros.

All symbols declared in a hygienic macro are unique.

In Common Lisp, hygiene is up to the programmer.

In Scheme, hygiene is provided by the macro facility.

Syntax-case

Recent scheme uses a `(syntax-case)` form for macros.

It combines two things:

- Simple syntactic transformations (with minimal computation)
- A generalized macro system as we saw in Common Lisp

It is hygienic by default.

Special operators can create non-hygienic macros.

Syntax-case example

(Almost) from the GNU Guile documentation, (`when`):

```
(define-syntax when
  (lambda (x)
    (syntax-case x ()
      ((when test e e* ...)
         (syntax
           (if test (begin e e* ...)))))))
```

Note the ellipses for repetition!

Destructuring

Destructuring allows declarations to describe structure:

```
(defmacro dolist ((var elts &optional retval) &body
   body)
  ;; expansion
  ))
```

Scheme also provides destructuring of repetition. [3]

```
(define-syntax my-let
  (lambda (x)
    (syntax-case x ()
      ((my-let ((var val) ...) body ...)
       (syntax
        ((lambda (var ...) body ...) val ...))))))
```

Destructuring in Other Contexts

Both Scheme and Common Lisp have general destructuring.

In Common Lisp, use `(destructuring-bind)`.

In Scheme, there are several macro packages.

Andrew Wright's pattern matcher [4] is in most implementations.

References I

Optional Readings

[1] Guy L. Steele Jr. *Common Lisp: The Language*. Second Edition. Digital Press, 1990.

[2] Eugene Kohlbecker et al. Tech. rep. 194. Indiana University, May 1986. URL: `https://legacy.cs.indiana.edu/ftp/techreports/TR194.pdf`.

[3] Guy L. Steele and Richard P. Gabriel. "The Evolution of Lisp". In: *History of Programming Languages II*. Association for Computing Machinery, 1996, pp. 233–330.

[4] Andrew K. Wright and Bruce F. Duba. *Pattern Matching for Scheme*. May 1995. URL: `https://citeseerx.ist.psu.edu/document?&doi=bb49b002a82cbd288ee5b0b9113b5f1ce8338a60`.

License