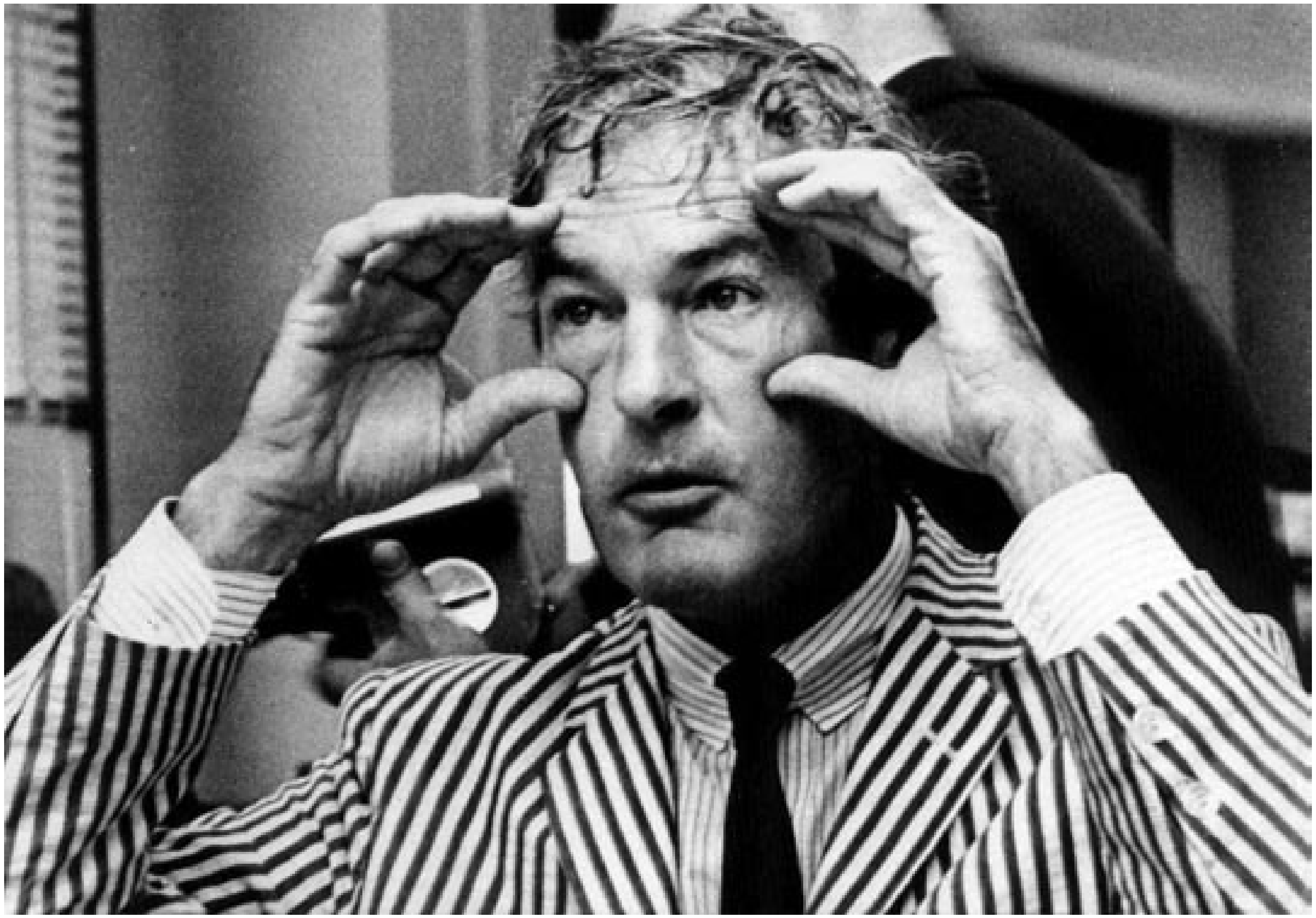
From Syntactic Sugar to the Syntactic Meth Lab:

Using Macros to Cook the Language You Want [a]

COMPUTER SCIENCE 21B: Structure and Interpretation of Computer Programs

[a] Drugs are bad, m'kay?

Wake up!



Sugar, Sugar

```
(define (mc-eval exp env) ;; the meta-circular evaluator with syntactic sugar.
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters exp)
                                       (lambda-body exp)
                                       env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((application? exp) (mc-apply (mc-eval (operator exp) env)
                                      (list-of-values (operands exp) env)))
        ;; Extra language features, via syntactic sugar.
        ((let? exp) (mc-eval (unsugar-let exp) env))
        ((cond? exp) (mc-eval (unsugar-cond exp) env))
        ((while? exp) (mc-eval (unsugar-while exp) env))
        ((for? exp) (mc-eval (unsugar-for exp) env))
        (else
          (error "Unknown expression type – EVAL" exp))))
```

Boring

We’re just doing the same thing over and over again. To add sugar, we:

Boring

We're just doing the same thing over and over again. To add sugar, we:

1. Create a new kind of expression (a list starting with unique symbol).
   $thing?$

Boring

We're just doing the same thing over and over again. To add sugar, we:

1. Create a new kind of expression (a list starting with unique symbol).
   thing?
2. Write a transformation (a Scheme procedure) from the new form into a simpler form.
   unsugar-thing

Boring

We're just doing the same thing over and over again. To add sugar, we:

1. Create a new kind of expression (a list starting with unique symbol).
   $thing?$
2. Write a transformation (a Scheme procedure) from the new form into a simpler form.
   $unsugar\text{-}thing$
3. Eval the new form by applying the transformation to obtain a simpler form and eval *that* form in the current environment.
   (($thing?\ exp$) ($mc\text{-}eval$ ($unsugar\text{-}thing\ exp$) env))

Transformations: let

Transformations (*unsugarings*) are just Scheme procedures that input a list and output a list (representing Scream expressions).

$$\boxed{\begin{array}{c}\text{'(let ((x e) }\cdots\text{) b)}\\ \Longrightarrow \\ \text{'((lambda (x }\cdots\text{) b) e }\cdots\text{)}\end{array}}$$

Transformations: let

Transformations (*unsugarings*) are just Scheme procedures that input a list and output a list (representing Scream expressions).

$$\boxed{\begin{array}{c}\text{'(let ((x e) }\cdots\text{) b)}\\ \Longrightarrow \\ \text{'((lambda (x }\cdots\text{) b) e }\cdots\text{)}\end{array}}$$

(define (*unsugar-let exp*)
  . . .)

Transformations: let

Transformations (*unsugarings*) are just Scheme procedures that input a list and output a list (representing Scream expressions).

$$\boxed{\begin{array}{c}\text{'(let ((x e) $\cdots$) b)} \\ \Longrightarrow \\ \text{'((lambda (x $\cdots$) b) e $\cdots$)}\end{array}}$$

```
(define (unsugar-let exp)
  (let ((xs (let-vars exp)) (es (let-exps exp)) (b  (let-body exp)))
    (cons (list 'lambda xs b) es)))
```

Transformations: let

Transformations (*unsugarings*) are just Scheme procedures that input a list and output a list (representing Scream expressions).

$$\boxed{\begin{array}{c}\text{'(let ((x e)} \cdots \text{) b)} \\ \Longrightarrow \\ \text{'((lambda (x} \cdots \text{) b) e} \cdots \text{)}\end{array}}$$

```
(define (unsugar-let exp)
  (let ((xs (let-vars exp)) (es (let-exps exp)) (b  (let-body exp)))
    (cons (list 'lambda xs b) es)))

(define (let-vars exp) (map car (cadr exp)))
(define (let-exps exp) (map cadr (cadr exp)))
(define (let-body exp) (caddr exp))
```

Transformations: cond

$$\boxed{\begin{array}{c}\text{'(cond (p1 e1) (p2 e2)} \cdots \text{(pn en))} \\ \Longrightarrow \\ \text{'(if p1 e1 (if p2 e2} \cdots \text{(if pn en false)} \cdots \text{))}\end{array}}$$

Transformations: cond

$$\boxed{\begin{array}{c}\text{'(cond (p1 e1) (p2 e2)} \cdots \text{(pn en))} \\ \Longrightarrow \\ \text{'(if p1 e1 (if p2 e2} \cdots \text{(if pn en false)} \cdots \text{))}\end{array}}$$

(define (*unsugar-cond exp*)
  . . .)

Transformations: cond

$$\boxed{\begin{array}{c}\text{'(cond (p1 e1) (p2 e2)} \cdots \text{(pn en))} \\ \Longrightarrow \\ \text{'(if p1 e1 (if p2 e2} \cdots \text{(if pn en false)} \cdots \text{))}\end{array}}$$

```
(define (unsugar-cond exp)
  (if (null? (cond-clauses exp))
      'false
      ...))
```

Transformations: cond

$$\boxed{\begin{array}{c}\text{'(cond (p1 e1) (p2 e2)} \cdots \text{(pn en))} \\ \Longrightarrow \\ \text{'(if p1 e1 (if p2 e2} \cdots \text{(if pn en false)} \cdots \text{))}\end{array}}$$

```
(define (unsugar-cond exp)
  (if (null? (cond-clauses exp))
      'false
      (list 'if
            (clause-predicate (car (cond-clauses exp)))
            (clause-expression (car (cond-clauses exp)))
            (cons 'cond (cdr (cond-clauses exp))))))
```

Transformations: cond

$$\boxed{\begin{array}{c}\text{'(cond (p1 e1) (p2 e2)} \cdots \text{(pn en))} \\ \Longrightarrow \\ \text{'(if p1 e1 (if p2 e2} \cdots \text{(if pn en false)} \cdots \text{))}\end{array}}$$

```
(define (unsugar-cond exp)
  (if (null? (cond-clauses exp))
      'false
      (list 'if
            (clause-predicate (car (cond-clauses exp)))
            (clause-expression (car (cond-clauses exp)))
            (cons 'cond (cdr (cond-clauses exp))))))

(define cond-clauses ...)
(define clause-predicate ...)
(define clause-expression ...)
```

Sugar (Refactored)

Let's refactor the code so *mc-eval* doesn't have to change with each new form of syntactic sugar:

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        . . .
        ((let? exp) (mc-eval (unsugar-let exp) env))
        ((cond? exp) (mc-eval (unsugar-cond exp) env))
        ((while? exp) (mc-eval (unsugar-while exp) env))
        ((for? exp) (mc-eval (unsugar-for exp) env))
        (else
         (error "Unknown expression type – EVAL" exp))))
```

Sugar (Refactored)

Let's refactor the code so *mc-eval* doesn't have to change with each new form of syntactic sugar:

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        . . .
        ((sugar? exp) (mc-eval (unsugar exp) env))
        (else
         (error "Unknown expression type – EVAL" exp))))
```

Sugar (Refactored)

- We need a table of sugar (name, transformation) pairs.

Sugar (Refactored)

- We need a table of sugar (name, transformation) pairs.

```
(define *sugar* (list (cons 'let unsugar-let)
                      (cons 'cond unsugar-cond) ... ))
```

Sugar (Refactored)

- We need a table of sugar (name, transformation) pairs.

```
(define *sugar* (list (cons 'let unsugar-let)
                      (cons 'cond unsugar-cond) ...))
```

- We need a *sugar?* predicate that returns true if the given expression starts with a name in the sugar table.

Sugar (Refactored)

- We need a table of sugar (name, transformation) pairs.

```
(define *sugar* (list (cons 'let unsugar-let)
                      (cons 'cond unsugar-cond) ...))
```

- We need a *sugar?* predicate that returns true if the given expression starts with a name in the sugar table.

```
(define (sugar? exp) (assq (car exp) *sugar*))
```

Sugar (Refactored)

- We need a table of sugar (name, transformation) pairs.

```
(define *sugar* (list (cons 'let unsugar-let)
                      (cons 'cond unsugar-cond) ...))
```

- We need a *sugar?* predicate that returns true if the given expression starts with a name in the sugar table.

```
(define (sugar? exp) (assq (car exp) *sugar*))
```

- We need *unsugar* which unsugars an expression according to the transformation in the table.

Sugar (Refactored)

- We need a table of sugar (name, transformation) pairs.

```
(define *sugar* (list (cons 'let unsugar-let)
                      (cons 'cond unsugar-cond) ...))
```

- We need a *sugar?* predicate that returns true if the given expression starts with a name in the sugar table.

```
(define (sugar? exp) (assq (car exp) *sugar*))
```

- We need *unsugar* which unsugars an expression according to the transformation in the table.

```
(define (unsugar exp)
  (let ((transform (cdr (assq (car exp) *sugar*))))
    (transform exp)))
```

Sugar (Refactored)

To add syntactic sugar, write an unsugar Scheme procedure and add an entry to the $*sugar*$ table.

(define $*sugar*$ ($list$ ($cons$ 'thing $unsguar\text{-}thing$) $\ldots$))

Sugar (Refactored)

To add syntactic sugar, write an unsugar Scheme procedure and add an entry to the $*sugar*$ table.

(**define** $*sugar*$ (*list* (*cons* 'thing *unsguar-thing*) . . .))

Let's write a procedure to make it easy:

Sugar (Refactored)

To add syntactic sugar, write an unsugar Scheme procedure and add an entry to the $*sugar*$ table.

(**define** $*sugar*$ (*list* (*cons* 'thing *unsguar-thing*) ...))

Let's write a procedure to make it easy:

```
(define (define-sugar! name transformation)
  (set! *sugar* (cons (cons name transformation) *sugar*)))
```

Sugar (Refactored)

To add syntactic sugar, write an unsugar Scheme procedure and add an entry to the *sugar* table.

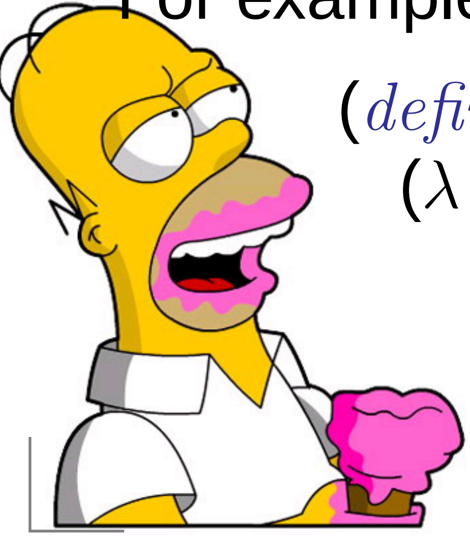
```
(define *sugar* (list (cons 'thing unsguar-thing) . . . ))
```

Let's write a procedure to make it easy:

```
(define (define-sugar! name transformation)
  (set! *sugar* (cons (cons name transformation) *sugar*)))
```

For example:

```
(define-sugar! 'delay
  (λ (exp) (list 'lambda '() (cadr exp))))
```



Sugar (Refactored)

To add syntactic sugar, write an unsugar Scheme procedure and add an entry to the *∗sugar∗* table.

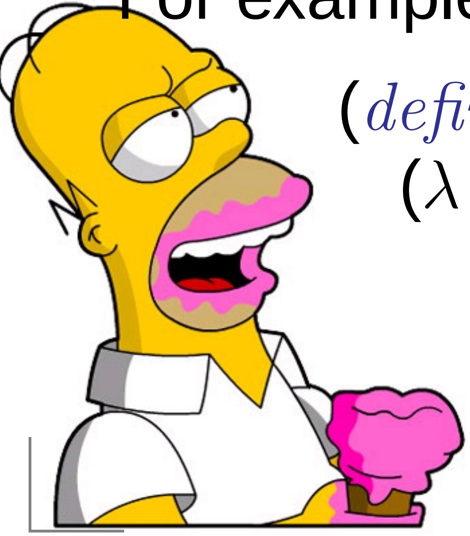
(define *∗sugar∗* (*list* (*cons* 'thing *unsguar-thing*) . . .))

Let's write a procedure to make it easy:

(define (*define-sugar! name transformation*)
  (set! *∗sugar∗* (*cons* (*cons name transformation*) *∗sugar∗*)))

For example:

(*define-sugar!* 'delay
  (λ (*exp*) (*list* 'lambda '() (*cadr exp*))))



We still haven't done anything interesting.

Macros

The idea: Why not give *define-sugar!* to the user? In other words: Make a define-sugar Scream form.

- As a language implementor, we only need to worry about the "core" forms.
- As a user, we can cook our own *syntactic abstractions*.

We'd like to write (in Scream):

```
(define-sugar delay
  (lambda (exp) (list 'lambda '() (cadr exp))))

(define (force thunk) (thunk))
```

. . . code using delay and force. . .

How would we do this?

How can we make *define-sugar* a Scream form?

```
(define (mc-eval exp env)
  (cond
    ...
    ((define-sugar? exp)  (define-sugar! (cadr exp)  (??? exp)) 'ok)
    ...))
```

How would we do this?

How can we make *define-sugar* a Scream form?

```
(define (mc-eval exp env)
  (cond
    ...
    ((define-sugar? exp)  (define-sugar! (cadr exp)  (??? exp)) 'ok)
    ...))
```

But what should *???* be? *caddr*?

```
(caddr '(define-sugar delay
              (lambda (exp) (list 'lambda '() (cadr exp)))))
```

$\Longrightarrow$ *?*

How would we do this?

How can we make *define-sugar* a Scream form?

```
(define (mc-eval exp env)
  (cond
    ...
    ((define-sugar? exp) (define-sugar! (cadr exp) (??? exp)) 'ok)
    ...))
```

But what should *???* be? *caddr*?

```
(caddr '(define-sugar delay
              (lambda (exp) (list 'lambda '() (cadr exp)))))
⟹ '(lambda (exp) (list 'lambda '() (cadr exp)))
```

How would we do this?

How can we make *define-sugar* a Scream form?

```
(define (mc-eval exp env)
  (cond
    . . .
    ((define-sugar? exp)  (define-sugar! (cadr exp)  (??? exp)) 'ok)
    . . .))
```

But what should *???* be? *caddr*?

```
(caddr '(define-sugar delay
              (lambda (exp) (list 'lambda '() (cadr exp)))))
⟹ '(lambda (exp) (list 'lambda '() (cadr exp)))
```

Will this work?

How would we do this?

How can we make *define-sugar* a Scream form?

```
(define (mc-eval exp env)
  (cond
    ...
    ((define-sugar? exp) (define-sugar! (cadr exp) (??? exp)) 'ok)
    ...))
```

But what should *???* be? *caddr*?

```
(caddr '(define-sugar delay
              (lambda (exp) (list 'lambda '() (cadr exp)))))
⟹ '(lambda (exp) (list 'lambda '() (cadr exp)))
```

Will this work?

```
(define-sugar! 'delay
   '(lambda (exp) (list 'lambda '() (cadr exp))))
```

The problem

(lambda (exp) (list 'lambda '() (cadr exp)))

$\neq$

(**lambda** (*exp*) (*list* 'lambda '() (*cadr exp*)))

- The *define-sugar!* procedure is expecting a name (a symbol) and a Scheme procedure.
- We're giving it a name and a list representing a (Scheme?, Scream?) procedure.

The problem

(lambda (exp) (list 'lambda '() (cadr exp)))

$$\neq$$

(**lambda** (*exp*) (*list* 'lambda '() (*cadr* *exp*)))

- The *define-sugar!* procedure is expecting a name (a symbol) and a Scheme procedure.
- We're giving it a name and a list representing a (Scheme?, Scream?) procedure.

- Are we *toadily screwed?*

The problem

(lambda (exp) (list 'lambda '() (cadr exp)))

$$\neq$$

(**lambda** (*exp*) (*list* 'lambda '() (*cadr exp*)))

- The *define-sugar!* procedure is expecting a name (a symbol) and a Scheme procedure.
- We're giving it a name and a list representing a (Scheme?, Scream?) procedure.

- Are we *toadily screwed?*
- *Which is it? Scheme? Scream?*

The problem

```
(lambda (exp) (list 'lambda '() (cadr exp)))
```

$$\neq$$

(lambda (*exp*) (*list* 'lambda '() (*cadr exp*)))

- The *define-sugar!* procedure is expecting a name (a symbol) and a Scheme procedure.
- We're giving it a name and a list representing a (Scheme?, Scream?) procedure.

- Are we *toadily screwed?*
- *Which is it? Scheme? Scream?*

"When you come to a fork in the road, take it."
– Yogi Berra

Transformations as Scheme procedures

If we want to implement transformations as Scheme procedures, we have to solve the problem of going from a list representation of a **lambda** expression, to the **lambda** expression itself.

(lambda (exp) (list 'lambda '() (cadr exp)))

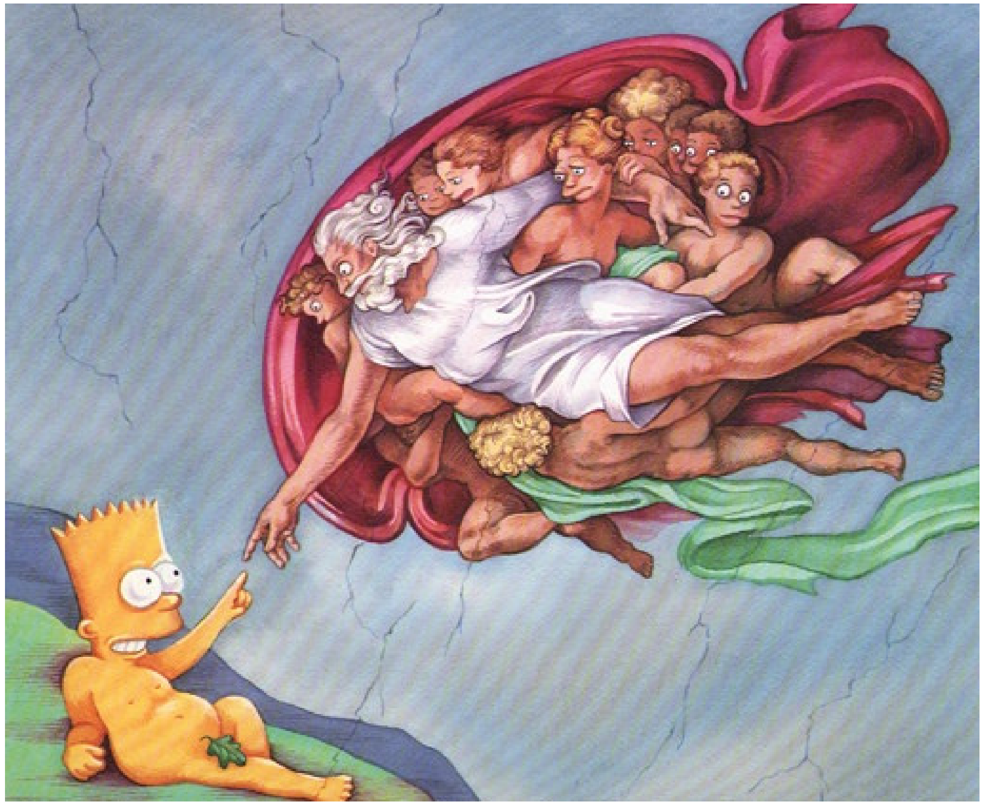
$$\Rightarrow$$

(**lambda** (*exp*) (*list* 'lambda '() (*cadr exp*)))

- Requires linguistic transubstantiation.
- As with any act of God—it should not be taken lightly.
- We have not seen how to do this in this course.

Scheme has its own meta-circular evaluator: eval.

eval



(*eval* '(lambda (exp) (list 'lambda '() (cadr exp))))

$\Rightarrow$

(**lambda** (*exp*) (*list* 'lambda '() (*cadr exp*)))

Sugar using *eval*

Let's look at *mc-eval* now:

```
(define (mc-eval exp env)
    . . .
    ((define-sugar? exp)
     (define-sugar! (sugar-name exp)
        (eval (sugar-transform exp)))
     'ok) · · ·)
```

```
(define sugar-name . . . )  ;; usual syntax stuff
(define sugar-transform . . . )
```

- We've successfully given the user the power of *define-sugar!*.
- We've broken a serious abstraction barrier.

Exposed!

- We started off trying to give the user the ability to specify expression transformations for syntactic sugar.
- We ended up *embedding the meta-language in the object-language*.
- Exposing the implementing language is *bad*.
- What if we wanted to change the implementing language? (To say, Haskell or C?)

Another consequence of this approach: transformations cannot refer to Scream values.

```
(define (unsugar-thing exp) . . . )
(define-sugar thing unsugar-thing)
```

Scheme knows nothing about unsugar-thing.

Transformations as Scream procedures

Let's rethink the define-sugar approach.

- If we can program list transformations in Scream, can't we use Scream procedures to implement sugar?

We can program things like the following in Scream after all, right?

```
(define (unsugar-let exp)
  (cons (list 'lambda (let-vars exp) (let-body exp)) (let-exps exp)))
```

```
(define (let-vars exp) (map car (cadr exp)))
(define (let-exps exp) (map cadr (cadr exp)))
(define (let-body exp) (caddr exp))
```

Transformations as Scream procedures

Let’s rethink the define-sugar approach.

- The $*sugar*$ table should still associated names with transformations, but transformations should now be Scream procedures. (let’s assume it’s initially empty now.)

A transformation is a Scream procedure that inputs a list and outputs a list (representing a Scream expression).

What’s shakin’?

- $sugar?$ unchanged.
- $define\text{-}sugar!$ unchanged.
- **(define** $*sugar*$ ’()).
- $unsugar$ has to change. Why?

Transformations as Scream procedures

Let's rethink the define-sugar approach.

- The old *unsugar* procedure looked up a transformation (a Scheme procedure) in **sugar**, then applied the procedure to the given expression.

```
(define (unsugar exp)
  (let ((transform (cdr (assq (car exp) *sugar*))))
    (transform exp)))
```

Now, lookup stays the same, but *transform* is now a Scream procedure. How can we apply it?

```
(define (unsugar exp)
  (let ((transform (cdr (assq (car exp) *sugar*))))
    ...))
```

Transformations as Scream procedures

Let's rethink the define-sugar approach.

- The old *unsugar* procedure looked up a transformation (a Scheme procedure) in *∗sugar∗*, then applied the procedure to the given expression.

```
(define (unsugar exp)
  (let ((transform (cdr (assq (car exp) *sugar*))))
    (transform exp)))
```

Now, lookup stays the same, but *transform* is now a Scream procedure. How can we apply it?

```
(define (unsugar exp)
  (let ((transform (cdr (assq (car exp) *sugar*))))
    (mc-apply transform (list exp))))
```

Transformations as Scream procedures

What do we know so far?

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        . . .
        ((sugar? exp) (mc-eval (unsugar exp) env))
        ((define-sugar? exp)
         (define-sugar! (sugar-name exp)
            . . . (sugar-transform exp) . . . )
         'ok)
        (else
         (error "Unknown expression type – EVAL" exp))))
```

Transformations as Scream procedures

How should we fill this out?

(*define-sugar!* (*sugar-name exp*)
  . . . (*sugar-transform exp*) . . .)

In English, how do we want to evaluate (define-sugar name exp)?

Transformations as Scream procedures

How should we fill this out?

$$(\textit{define-sugar!}\ (\textit{sugar-name exp})$$
$$\ldots\ (\textit{sugar-transform exp})\ \ldots)$$

In English, how do we want to evaluate (define-sugar name exp)?

- Evaluate exp.

Transformations as Scream procedures

How should we fill this out?

$$\begin{array}{l}(\textit{define-sugar!}\ (\textit{sugar-name exp}) \\ \quad \ldots\ (\textit{sugar-transform exp})\ \ldots)\end{array}$$

In English, how do we want to evaluate (define-sugar name exp)?

- Evaluate exp. In what environment?

Transformations as Scream procedures

How should we fill this out?

(*define-sugar!* (*sugar-name exp*)
 . . . (*sugar-transform exp*) . . .)

In English, how do we want to evaluate (define-sugar name exp)?

- Evaluate exp. In what environment? The current environment.

Transformations as Scream procedures

How should we fill this out?

$(\textit{define-sugar!}\ (\textit{sugar-name exp})$
$\quad \ldots\ (\textit{sugar-transform exp})\ \ldots)$

In English, how do we want to evaluate (define-sugar name exp)?

- Evaluate exp. In what environment? The current environment.
- It should be a procedure. Associate name with this procedure in the table of syntactic sugar forms.

Transformations as Scream procedures

How should we fill this out?

$$(\textit{define-sugar!}\ (\textit{sugar-name exp})$$
$$\ldots\ (\textit{sugar-transform exp})\ \ldots)$$

In English, how do we want to evaluate (define-sugar name exp)?

- Evaluate exp. In what environment? The current environment.
- It should be a procedure. Associate name with this procedure in the table of syntactic sugar forms.

$$(\textit{define-sugar!}\ (\textit{sugar-name exp})$$
$$(\textit{mc-eval}\ (\textit{sugar-transform exp})\ \textit{env}))$$

Done(?)

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        . . .
        ((sugar? exp) (mc-eval (unsugar exp) env))
        ((define-sugar? exp)
         (define-sugar! (sugar-name exp)
              (mc-eval (sugar-transform exp) env))
         'ok)
        . . .
        (else
         (error "Unknown expression type – EVAL" exp))))
```

```
(define (unsugar exp)
   (let ((transform (cdr (assq (car exp) *sugar*))))
      (mc-apply transform (list exp))))

(define *sugar* '())
(define (sugar? exp) (assq (car exp) *sugar*))
(define (define-sugar! name transformation)
   (set! *sugar* (cons (cons name transformation) *sugar*)))
```

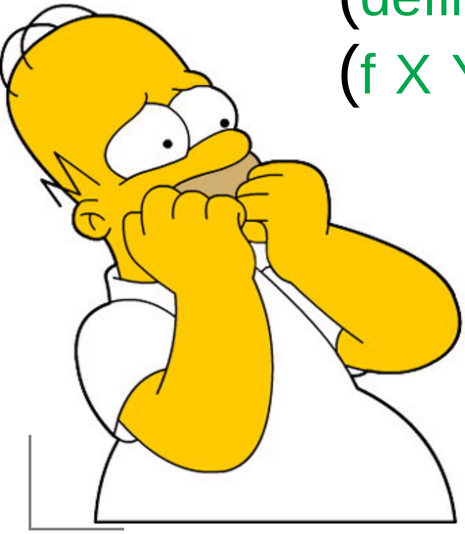
Some worrisome examples

- Application or Sugar?

```
(define f ...)
(define-sugar f ...)
(f X Y Z)
```

- What about?

```
(define-sugar f ...)
(define f ...)
(f X Y Z)
```



- What about?

```
(define-sugar f ...)
(let ((f ...)) (f X Y Z))
```

- What should happen here?

```
(define-sugar f ...)
(define g ...)
(g f)
```

Analogous to (g **cond**) in Scheme.

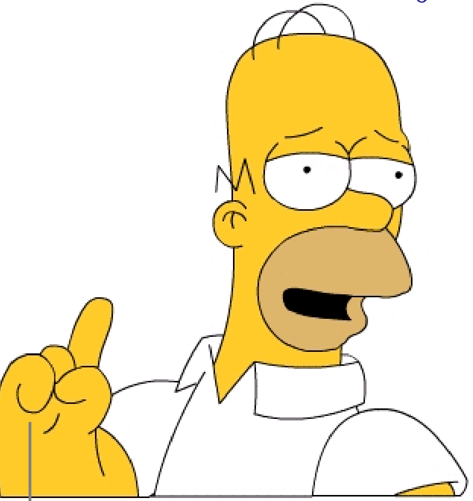
The problem

- We have essentially two environment structures. One for (variable,value) pairs. One for (sugar name,transformer) pairs.
- By making them distinct sugar names aren't properly shadowed.

Why not merge the two environment structures into one? In other words, let's put $*sugar*$ into env.

The solution (sketch)

- Change environment structure to include (('sugar *var*) *transform*) associations.
- *lookup-variable-value* has to change.
- *sugar?* should check for a ('sugar *var*) binding in *env*.
- *unsugar* should get the *transform* from *env*.
- *define-sugar!* should take an *env* and update it (much like *define-variable!*).



- If we lookup a variable's value and find it is bound to a sugar transformation, raise an error, eg. (f cond).

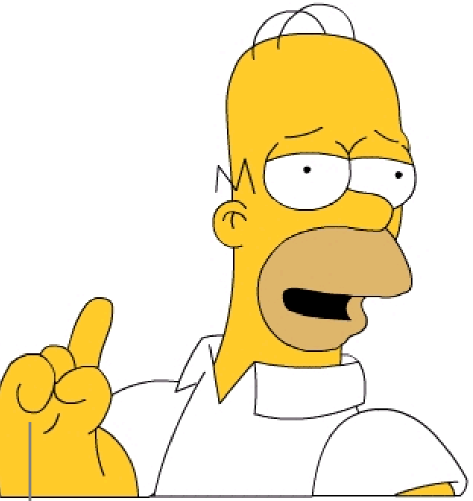
You can do all of these in the privacy of your own home.

let-sugar

We can now add new constructs:

```
(let-sugar ((f . . . )
            (g . . . ))
  . . .  code using f and g . . . )
```

```
(define (mc-eval exp env)
  . . .
  ((let-sugar? exp)
   (mc-eval (let-sugar-body exp)
            (extend-environment
              (map (λ (name) (list 'sugar name))
                   (let-sugar-names exp))
              (let-sugar-transforms exp)
              env)))
  . . .)
```



Some examples

- Delays and Streams:

```
(define-sugar delay
  (lambda (exp) (list 'lambda '() (cadr exp))))

(define (force thunk) (thunk))

(define-sugar stream-cons
  (lambda (exp) (list 'cons (cadr exp) (list 'delay (caddr exp)))))

(define the-empty-stream '())
(define empty-stream? null?)

(define stream-car car)
(define (stream-cdr s) (force (cdr s)))
```

Some examples

- All the sugar you can eat:

```
(define-sugar let
  (lambda (exp)
    (cons (list 'lambda (let-vars exp) (let-body exp)) (let-exps exp))))

(define-sugar cond
  (lambda (exp)
    (if (null? (cond-clauses exp))
        'false
        (list 'if
              (clause-predicate (car (cond-clauses exp)))
              (clause-expression (car (cond-clauses exp)))
              (cons 'cond (cdr (cond-clauses exp)))))))
```

Some ideas

- Little languages:

```
(define-sugar regexp
  (lambda (exp) . . . ))

(define starts-with-xys-or-pqs?
  (regexp (| (+ (| "x" "y"))
             (+ (| "p" "q")))))

(define-sugar automaton
  (lambda (exp) . . . ))
```

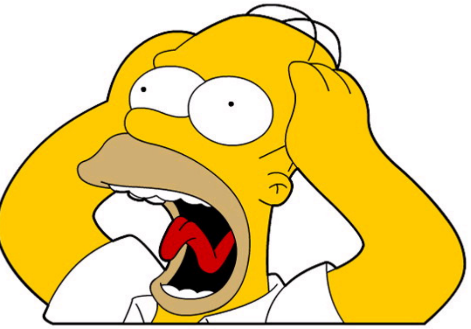
```
(define m
  (automaton init
    (init : (c → more))
    (more : (a → more)
            (d → more)
            (r → end))
    (end : accept)))

(m '(c a d d a r)) ⇒ true
(m '(c a d d a r r)) ⇒ false
```

The hygiene problem

Consider the following swap! macro:

```
(define-sugar swap!
  (lambda (exp)
   (list 'let (list (list 'tmp (cadr exp)))
      (list 'set! (cadr exp) (caddr exp))
      (list 'set! (caddr exp) 'tmp))))
```



```
(let ((x 1) (y 2))
  (swap! x y))
⇒
(let ((x 1) (y 2))
  (let ((tmp x))
    (set! x y)
    (set! y tmp)))
```

```
(let ((tmp 1) (y 2))
  (swap! tmp y))
⇒
(let ((tmp 1) (y 2))
  (let ((tmp tmp))
    (set! tmp y)
    (set! y tmp)))
```

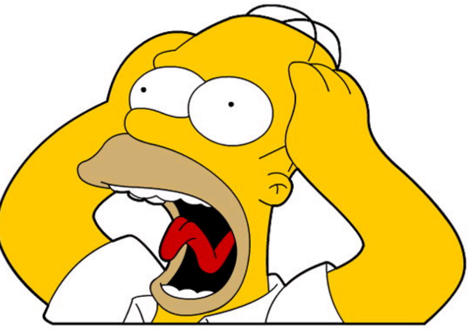
The hygiene problem

Consider the following stream-cons! macro:

```
(define-sugar stream-cons
  (lambda (exp) (list 'cons (cadr exp) (list 'delay (caddr exp)))))
```

```
(stream-cons 1 null-stream)
⇒
(cons 1 (delay null-stream))
```

```
(let ((cons 1))
  (stream-cons 1 null-stream))
⇒
(let ((cons 1))
  (cons 1 (delay null-stream)))
```



Scheme Macros

There are two flavors of Scheme macros.

- A “procedural” system much like what we’ve just seen, except it solves the hygiene problem and uses a syntax datatype in place of lists and symbols.
- A “rewriting” system that uses pattern matching and templates to specify transformations (also hygienic).

Let’s look at the rewriting system (aka **syntax-rules**).

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body)
     ((lambda (name ...) body) val ...))))
```

Scheme Macros

```
(define-syntax delay
  (syntax-rules ()
    ((delay e)
     (lambda () e))))

(define-syntax stream-cons
  (syntax-rules ()
    ((stream-cons x y)
     (cons x (delay y)))))
```

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 . . . )
     (if test1 (and test2 . . . ) #f))))

(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 . . . )
     (let ((x test1))
       (if x x (or test2 . . . ))))))
```

Scheme Macros

```
(define-syntax automaton
  (syntax-rules (:)
   ((automaton init-state (state : response . . . ) . . . )
    (let-syntax ((process-state (syntax-rules (accept →)
                                  ((_ accept)
                                   (lambda (input)
                                     (empty? input)))
                                  ((_ (label → target) (. . .   . . . ))
                                   (lambda (stream)
                                     (case (first input)
                                       ((label) (target (rest input)))
                                       (. . .   . . . )
                                       (else #f)))))))
        (letrec ((state (process-state response . . . )) . . . )
          init-state)))))
```

Head