From Syntactic Sugar to the Syntactic Meth Lab:

*Using Macros to Cook the Language You Want*

*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-quotational 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:
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). 
   \textit{thing}?

2. Write a transformation (a \textit{Scheme} procedure) from the new form into a simpler form. 
   \textit{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).
   \[ \text{thing?} \]

2. Write a transformation (a Scheme procedure) from the new form into a simpler form.
   \[ \text{unsugar-thing} \]

3. Eval the new form by applying the transformation to obtain a simpler form and eval \textit{that} form in the current environment.
   \[ ((\text{thing? exp}) (\text{mc-eval (unsugar-thing exp) env})) \]
Transformations (unsugarings) are just Scheme procedures that input a list and output a list (representing Scream expressions).

\[
\text{'(let ((x e) \ldots) b)} \\
\quad \Rightarrow \\
\text{'((lambda (x \ldots) b) e \ldots)}
\]
Transformations: let

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

\[
\text{'(let ((x e) \ldots) b)} \Rightarrow \text{'((lambda (x \ldots) b) e \ldots)}
\]

(define (unsugar-let exp)
  \ldots)
Transformations: let

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

\[
\text{\'(let ((x e) \cdots) b)} \\
\Rightarrow \\
\text{\'((lambda (x \cdots) b) e \cdots)}
\]

(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).

\[
\begin{align*}
\text{'(let ((x e) \ldots) b)} & \quad \Rightarrow \\
\text{'(\((\lambda (x \ldots) b)\ e \ldots))}
\end{align*}
\]

```scheme
(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))
```

From Syntactic Sugar to the Syntactic Meth Lab: – p.5/35
Transformations: cond

\[
'(\text{cond} \ (p_1 \ e_1) \ (p_2 \ e_2) \ \cdots \ (p_n \ e_n))
\]

\[
\Rightarrow
\]

\[
'(\text{if} \ p_1 \ e_1 \ (\text{if} \ p_2 \ e_2 \ \cdots \ (\text{if} \ p_n \ e_n \ \text{false}) \ \cdots))
\]

From Syntactic Sugar to the Syntactic Meth Lab: – p.6/35
Transformations: cond

\[
'(\text{cond } (p_1 \ e_1) \ (p_2 \ e_2) \ \cdots \ (p_n \ e_n))
\]
\[
\Rightarrow
\]
\[
'(\text{if } p_1 \ e_1 \ (\text{if } p_2 \ e_2 \ \cdots \ (\text{if } p_n \ e_n \ \text{false}) \ \cdots))
\]

(define (unsugar-cond exp)
  \ldots)
Transformations: cond

\[
\left(\text{cond } (p_1 \; e_1) \; (p_2 \; e_2) \; \cdots \; (p_n \; e_n)\right)
\]

\[
\Rightarrow
\left(\text{if } p_1 \; e_1 \; (\text{if } p_2 \; e_2 \; \cdots \; (\text{if } p_n \; e_n \; \text{false}) \; \cdots)\right)
\]

\[
\text{(define} \; (\text{unsugar-cond } \text{exp})
\]

\[
\text{(if} \; (\text{null?} \; (\text{cond-\text{clauses } exp}))
\]

\[
\text{false
\]

\[
\cdots)\))
\]
Transformations: cond

\[(\text{cond} \ (p_1 \ e_1) \ (p_2 \ e_2) \ \cdots \ (p_n \ e_n)) \Rightarrow (\text{if} \ p_1 \ e_1\ (\text{if} \ p_2 \ e_2 \ \cdots \ (\text{if} \ p_n \ e_n \ \text{false} \ \cdots))\]

(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

\[
'(\text{cond } (p_1\ e_1)\ (p_2\ e_2)\ \cdots\ (p_n\ e_n))
\]
\[\Rightarrow\]
\[
'(\text{if } p_1\ e_1\ (\text{if } p_2\ e_2\ \cdots\ (\text{if } p_n\ e_n\ \text{false})\ \cdots)\)
\]

\begin{verbatim}
(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 \ldots)
(define clause-predicate \ldots)
(define clause-expression \ldots)
\end{verbatim}
Sugar (Refactored)

Let’s refactor the code so \textit{mc-eval} doesn’t have to change with each new form of syntactic sugar:

\begin{verbatim}
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        \ldots
        ((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))))
\end{verbatim}
Sugar (Refactored)

Let’s refactor the code so \textit{mc-eval} doesn’t have to change with each new form of syntactic sugar:

\begin{verbatim}
(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))))
\end{verbatim}
Sugar (Refactored)

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

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

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

From Syntactic Sugar to the Syntactic Meth Lab: – p.8/35
We need a table of sugar (name, transformation) pairs.

\[
\text{(define } \ast\text{sugar}\ast \ (\text{list } (\text{cons } \text{'let} \ \text{unsugar-let}) \\
\quad (\text{cons } \text{'cond} \ \text{unsugar-cond}) \ldots))
\]

We need a \text{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.

\[
(\text{define } *\text{sugar}* \ (\text{list } (\text{cons } \text{'let} \ \text{unsugar-let})
\ (\text{cons } \text{'cond} \ \text{unsugar-cond}) \ldots ))
\]

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

\[
(\text{define } (\text{sugar? exp}) \ (\text{assq (car exp) } *\text{sugar}*))
\]
We need a table of sugar (name, transformation) pairs.

\[
\text{(define } \ast\text{sugar}\ast\text{ } (\text{list } (\text{cons 'let unsugar-let})
\text{ (cons 'cond unsugar-cond) . . . }))
\]

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

\[
\text{(define } (\text{sugar? exp) (assq (car exp) } \ast\text{sugar}\ast\text{))}
\]

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.

  \[
  \text{(define } \ast \text{sugar} \ast \text{ (list (cons ’let unsugar-let)}
  \text{ (cons ’cond unsugar-cond) \ldots )))
  \]

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

  \[
  \text{(define (sugar? exp) (assq (car exp) \ast \text{sugar} \ast ))}
  \]

- We need \text{unsugar} which unsugars an expression according to the transformation in the table.

  \[
  \text{(define (unsugar exp)}
  \text{ (let ((transform (cdr (assq (car exp) \ast \text{sugar} \ast ))))}
  \text{ (transform exp)))}
  \]
Sugar (Refactored)

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

\[\text{(define *sugar* (list (cons 'thing unsugar-thing) \ldots))}\]
Sugar (Refactored)

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

\[
\text{(define } \textit{*sugar* (list (cons ’thing unsguar-thing) . . .))}
\]

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 \( *\text{sugar}* \) table.

\[
\text{(define } *\text{sugar}* \ (\text{list } (\text{cons ’thing unsguar-thing }) \ldots ))
\]

Let’s write a procedure to make it easy:

\[
\text{(define } (\text{define-sugar! name transformation}) \ \\
(\text{set! } *\text{sugar}* \ (\text{cons } (\text{cons name transformation} \ *\text{sugar}*))))
\]
Sugar (Refactored)

To add syntactic sugar, write an unsugar Scheme procedure and add an entry to the \textit{sugar} table.

\begin{verbatim}
(define *sugar* (list (cons ’thing unsugar-thing) . . . ))
\end{verbatim}

Let’s write a procedure to make it easy:

\begin{verbatim}
(define (define-sugar! name transformation)
  (set! *sugar* (cons (cons name transformation) *sugar*)))
\end{verbatim}

For example:

\begin{verbatim}
(define-sugar! ’delay
  (λ (exp) (list ’lambda ’() (cadr exp))))
\end{verbatim}
Sugar (Refactored)

To add syntactic sugar, write an unsugar Scheme procedure and add an entry to the \emph{sugar} table.

\begin{verbatim}
(define *sugar* (list (cons ’thing unsugar-thing) . . . ))
\end{verbatim}

Let’s write a procedure to make it easy:

\begin{verbatim}
(define (define-sugar! name transformation)
  (set! *sugar* (cons (cons name transformation) *sugar*)))
\end{verbatim}

For example:

\begin{verbatim}
(define-sugar! ’delay
  (λ (exp) (list ’lambda ’() (cadr exp))))
\end{verbatim}

We still haven’t done anything interesting.
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):

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

…code using delay and force…

From Syntactic Sugar to the Syntactic Meth Lab: -- p.10/35
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?

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

But what should `???` be? `caddr`?

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

From Syntactic Sugar to the Syntactic Meth Lab: – p.11/35
How would we do this?

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

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

But what should `???` be? `caddr`?

```scheme
(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?

\[
(\text{define} \ (mc\text{-eval} \ exp \ env) \\
(\text{cond} \\
\ldots \\
((\text{define-sugar?} \ exp) \ (\text{define-sugar!} \ (\text{cadr} \ exp) \ (??? \ exp)) \ 'ok) \ \\
\ldots)))
\]

But what should \texttt{???} be? `caddr`?

\[
(caddr \ '(\text{define-sugar} \ \text{delay} \\
(\lambda \ (exp) \ (\text{list} \ '\lambda \ () \ (\text{cadr} \ exp))))))
\]

\[
\Rightarrow \ '(\lambda \ (exp) \ (\text{list} \ '\lambda \ () \ (\text{cadr} \ exp)))
\]

Will this work?

\[
(\text{define-sugar!} \ '\text{delay} \\
'(\lambda \ (exp) \ (\text{list} \ '\lambda \ () \ (\text{cadr} \ exp))))
\]
The problem

\[(\textit{lambda} (\textit{exp}) (\textit{list} \textit{\textquote{lambda}} () (\textit{cad} \textit{exp}))) \neq (\textit{lambda} (\textit{exp}) (\textit{list} \textit{\textquote{lambda}} () (\textit{cad} \textit{exp})))\]

- The \textit{define-sugar!} procedure is expecting a name (a symbol) and a \texttt{Scheme} procedure.
- We’re giving it a name and a \texttt{list} representing a \texttt{(Scheme?, Scream?)} procedure.
The problem

\[ \text{(lambda (exp) (list 'lambda '() (cadr exp)))} \]
\[ \neq \]
\[ \text{(lambda (exp) (list 'lambda '() (cadr exp)))} \]

- The \textit{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 \textit{toadily screwed}?
The problem

\[(\lambda (exp) (\text{list } \lambda () (\text{cadr exp}))) \neq (\lambda (exp) (\text{list } \lambda () (\text{cadr exp})))\]

- The \textit{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 \textit{toadily screwed}?
- \textit{Which is it? Scheme? Scream?}
The problem

\[(\text{lambda} (\exp) (\text{list} \:\text{'lambda} () (\text{cadr} \exp)))\]

\[\neq\]

\[(\text{lambda} (\exp) (\text{list} \:\text{'lambda} () (\text{cadr} \exp)))\]

- 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
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.

\[
\text{(lambda (exp) (list 'lambda () (cadr exp)))}
\]

\[
\Rightarrow
\]

\[
\text{(lambda (exp) (list 'lambda () (cadr exp)))}
\]

- 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.
\[
\text{eval}\left(\text{eval}\left(\lambda (\text{exp}) \left(\text{list } \lambda () \left(\text{cadr exp}\right)\right)\right)\right)
\]

\[
\Rightarrow
\left(\lambda (\text{exp}) \left(\text{list } \lambda () \left(\text{cadr exp}\right)\right)\right)
\]
Let’s look at \textit{mc-eval} now:

\begin{verbatim}
(define (mc-eval exp env)
  ...
  ((define-sugar? exp)
   (define-sugar! (sugar-name exp)
     (eval (sugar-transform exp)))
   'ok) ...
)
\end{verbatim}

\begin{verbatim}
(define sugar-name ...);; usual syntax stuff
(define sugar-transform ...)
\end{verbatim}

\begin{itemize}
  \item We’ve successfully given the user the power of \textit{define-sugar}!
  \item We’ve broken a serious abstraction barrier.
\end{itemize}
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 associate 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-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.

\[
(\text{define} \ (\text{unsugar} \ \text{exp}) \\
(\text{let} \ ((\text{transform} \ (\text{cdr} \ (\text{assq} \ (\text{car} \ \text{exp}) *\text{sugar}*))))))
\]

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

\[
(\text{define} \ (\text{unsugar} \ \text{exp}) \\
(\text{let} \ ((\text{transform} \ (\text{cdr} \ (\text{assq} \ (\text{car} \ \text{exp}) *\text{sugar}*))))))
\]
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?

\[
\text{(define-sugar! (sugar-name exp) }
\text{ ... (sugar-transform exp) ...)}
\]

In English, how do we want to evaluate \((\text{define-sugar name exp})\)?
Transformations as Scream procedures

How should we fill this out?

\[
(\text{define-sugar!} (\text{sugar-name exp}) \\
\ldots (\text{sugar-transform exp}) \ldots)
\]

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

Evaluate \(\text{exp}\).
Transformations as Scream procedures

How should we fill this out?

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

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

Evaluate \text{exp}. In what environment?
Transformations as **Scream procedures**

How should we fill this out?

\[
\text{(define-sugar! (sugar-name exp) }
\ldots \text{ (sugar-transform exp) \ldots )}
\]

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

- Evaluate \(\text{exp}\). In what environment? The current 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.

- 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?

\[
(\text{define-sugar!} (\text{sugar-name} \ exp) \\
\ldots \ (\text{sugar-transform} \ exp) \ldots )
\]

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

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

It should be a procedure. Associate \text{name} with this procedure in the table of syntactic sugar forms.

\[
(\text{define-sugar!} (\text{sugar-name} \ exp) \\
(\text{mc-eval} (\text{sugar-transform} \ exp) \ env))
\]

From Syntactic Sugar to the Syntactic Meth Lab: – p.21/35
(define (mc-eval exp env)
  (cond (((self-evaluating? exp) 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*)))

From Syntactic Sugar to the Syntactic Meth Lab: – p.22/35
Some worrisome examples

- Application or Sugar?
  ```scheme```
  (define f ...)
  (define-sugar f ...)
  (f X Y Z)
  ```

- What about?
  ```scheme```
  (define-sugar f ...)
  (let ((f ...)) (f X Y Z))
  ```

- What should happen here?
  ```scheme```
  (define-sugar f ...)
  (define g ...)
  (g f)
  ```

Analogous to (g cond) in Scheme.

From Syntactic Sugar to the Syntactic Meth Lab: – p.23/35
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.
We can now add new constructs:

\[
\text{(let-sugar ((f . . .) (g . . .)) . . . code using f and g . . .)}
\]

\[
(\text{define (mc-eval exp env)}
\]

\[
\text{. . .}
\]

\[
((\text{let-sugar? exp})
\]

\[
(\text{mc-eval (let-sugar-body exp)}
\]

\[
(\text{extend-environment}
\]

\[
(\text{map (λ (name) (list 'sugar name)) (let-sugar-names exp))}
\]

\[
(\text{let-sugar-transforms exp) env}))
\]

\[
. . .
\]
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:

```lisp
(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:

\[
\text{(define-sugar stream-cons}
\begin{align*}
& \text{(lambda (exp) (list ’cons (cadr exp) (list ’delay (caddr exp))))})
\end{align*}
\]

\[
\text{(stream-cons 1 null-stream)} \Rightarrow (\text{cons 1 (delay null-stream)})
\]

\[
\text{(let ((cons 1))}
\begin{align*}
& \text{(stream-cons 1 null-stream)}
\end{align*}
\]

\[
\Rightarrow (\text{let ((cons 1))}
\begin{align*}
& \text{(cons 1 (delay null-stream))}
\end{align*}
\]
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 . )))
)```

From Syntactic Sugar to the Syntactic Meth Lab: – p.32/35
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 ...)))))

From Syntactic Sugar to the Syntactic Meth Lab: – p.33/35
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