A Basic Object System Using Macros

A Talk With Grant Rettke

grettke@acm.org

http://www.wisdomandwonder.com/

How To Roll A Basic Object System

- Features
  - Public Methods, Private Variables, No Inheritance (Simple huh?!)
- Goals
  - Study Scheme, Macros, Language Constructs
  - Chat About It With My Friends
- Approach
  - Code First, Then Generation
  - Simplicity Trumps Efficiency
  - No Mystery Code!

Destination Code Features

- Primitive Object Implementation
- Methods and Variables
- Encapsulation
- Message Passing
- Duplicate variable/method name warning
- Built on "Stock" Language Features
- Reference: 5-prim-obj-stx-smpl.scm, 6-prim-obj-stx-smpl-tsts.scm

Destination Code Sample

```
(define-object person
  (variables
   ([name #f]
    [age-years #f]
    [method-names-ls2 10]))
  (methods
   ([set-name [arg] (set! name arg)]
    [get-name [] name]
    [set-age-years [arg] (set! age-years arg)]
    [get-age-years [] age-years]
    [age-in-days [] (* age-years 365)]
    [typed-name [] (cons (get-name) (class-name))]
    [method-names2 [] 'OVERWRITTEN]
    [typed-name2 [] 'OVERWRITTEN]))))
```

- Brackets may be used anywhere parentheses are used

  - Primarily to enhance readability

Step 1

- Exploring Primitive Language Features
  - Object Creation
  - Message Passing
  - Lexical Scope
- Reference: 1-prm-feat.scm

Object Creation

```
(define prim-obj-creation
  (λ ()
    (λ ()
      #t)))
```

- Goal: An object is a thing that can be instantiated
- This code is a function that returns a 1st class function
- A 1st class function is a thing, virtually an "object"
- This is how objects will be instantiated in this system

Message Passing

```
(define prim-obj-msg-passing
  (λ ()
    (λ (msg)
      msg)))
```

- Goal: An object is a thing that can receive a message
- The first class function above can receive messages
- This is how "message-passing" will occur in this object system

Lexical Scope

```
(define prim-obj-lex-scope
  (λ ()
    (define x 11)
    (define y 12)
    (define frobnicate
      (λ (a b)
        (+ a b)))
    (λ (msg)
      (cons msg (frobnicate x y)))))
```

- Goal: Encapsulation, introduction of scope
- Lambda introduces lexical scope for internal **`define`** (aka **`letrec`**) appearing immediately after it
- The 1st class function returned by this function inherits the lexical scope in which it was created (**`x`**, **`y`**, and **`frobnicate`**)

prim-obj-lex-scope usage

```
(define obj (prim-obj-lex-scope))
(obj 'my-message)
```

- -> `(my-message . 23)`
- This is how objects may be instantiated and sent messages

The Non-Macro Primitive Object

- Combine those three primitive features to hand-code a primitive object
- Reference: 3-A-prim-obj.scm

The Non-Macro Primitive Object Code Sample

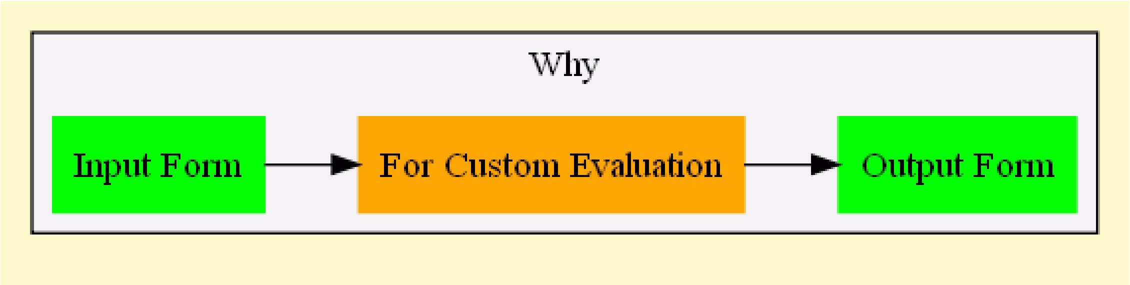
```
(define person
  (λ ()
    (define name #f)
    (define age-years #f)
    (define set-name (λ (arg) (set! name arg)))
    (define get-name (λ () name))
    <methods go here>
    (λ (msg . args)
      (case msg
        [(set-name) (apply set-name args)]
        [(get-name) (get-name)]
        [(set-age-years) (apply set-age-years args)]
        [(get-age-years) (get-age-years)]
        [(age-in-days) (age-in-days)]
        [else (error "message not understood" msg)])))
```

- The last λ expression is the "object"

Next Step: On To Generation

- High Level Macro Review

Macros 1 - Why



- Modify input code to produce new output code
- Seemingly superior to C style pre-processor macros
- Change shape, and even order of evaluation of, the code

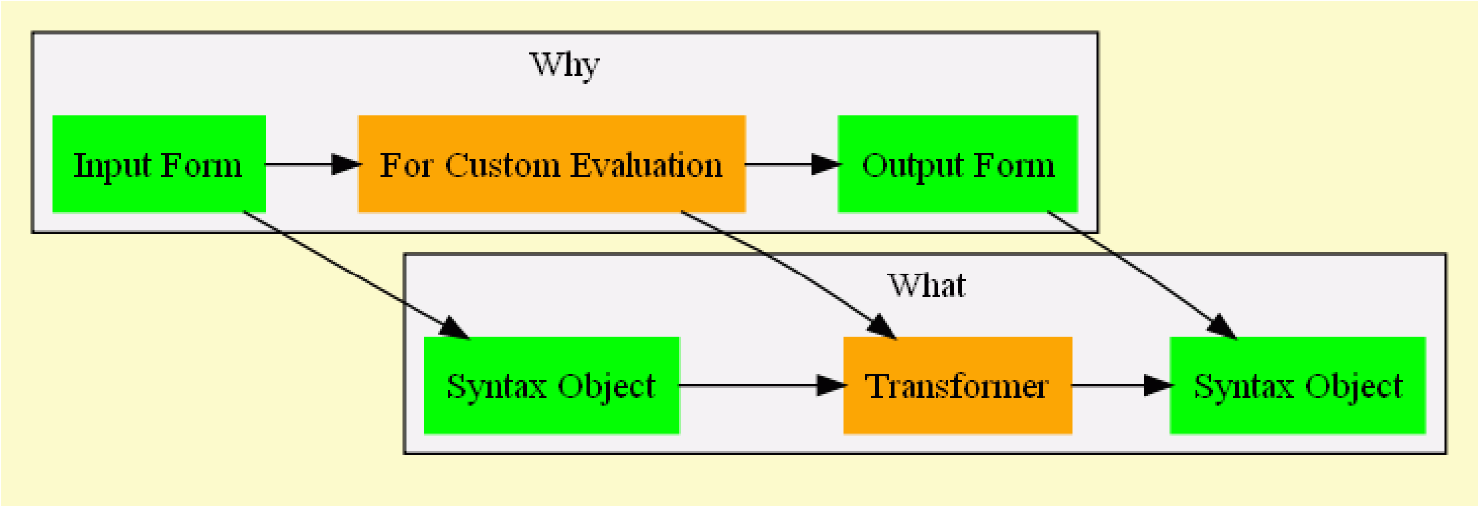
Macros 1 - Why - Example

- In a conditional expression, every clause may not be evaluated
- Consider a typical if-null-then check (illustrated by the macro **my-if** moving forward)

```
(let ([fun null])
  (my-if (null? fun)
       (printf "Can't call fun, it is null~n")
       (printf "x is ~a~n" (fun))))
```

- **my-if** could never be a function because it would evaluate its arguments, resulting in a null pointer
- Reference: macros.scm

Macros 2 - What



- The object sent to the macro is called a syntax object
- The macro itself is implemented by an object called a transformer

Macros 2 - What - Example

- Syntax Object (everything enclosed by **my-if**)

```
(let ([fun null])
  (my-if (null? fun)
         (printf "Can't call fun, it is null~n")
         (printf "x is ~a~n" (fun))))
```

- Transformer (this is the **my-if** macro)

```
[(_ clause true-body false-body)
 #'(let ([c clause])
     (if c
         true-body
         false-body))]
```

- #' is shorthand for surrounding the following shape inside a call to **syntax**

Macros 3 - How

- The macro **`my-if`** takes an input form as its argument

```
(my-if (null? fun)
       (printf "Can't call fun, it is null~n")
       (printf "x is ~a~n" (fun)))
```

- De-structures it using pattern matching into 3 different parts: **`clause`**, **`true-body`**, and **`false-body`**

```
(my-if clause true-body false-body)
```

- Defines a template for the new form (the resulting syntax object)

```
#'(let ([c clause]) (if c true-body false-body))
```

Macros 3 - How

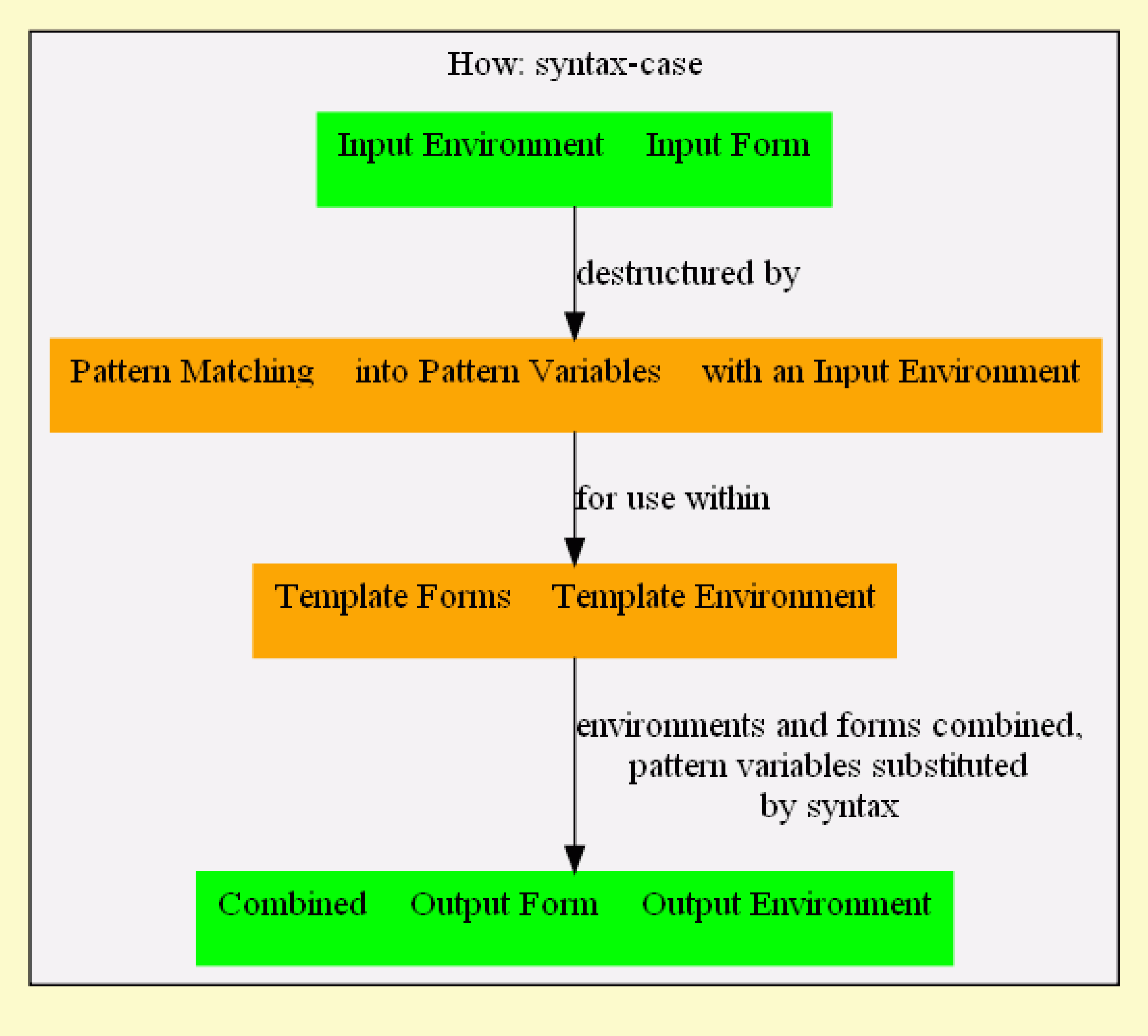
- Expands the template by replacing the pattern variables with their actual values and environment, and returns the resulting syntax-object

```
(my-if (null? fun)
       (printf "Can't call fun, it is null~n")
       (printf "x is ~a~n" (fun)))
```

- Expanding into

```
(let ((c (null? fun)))
  (if c
      (printf "Can't call fun, it is null~n")
      (printf "x is ~a~n" (fun))))
```

Macros 3 - How - Visual



Two Kinds of Macros

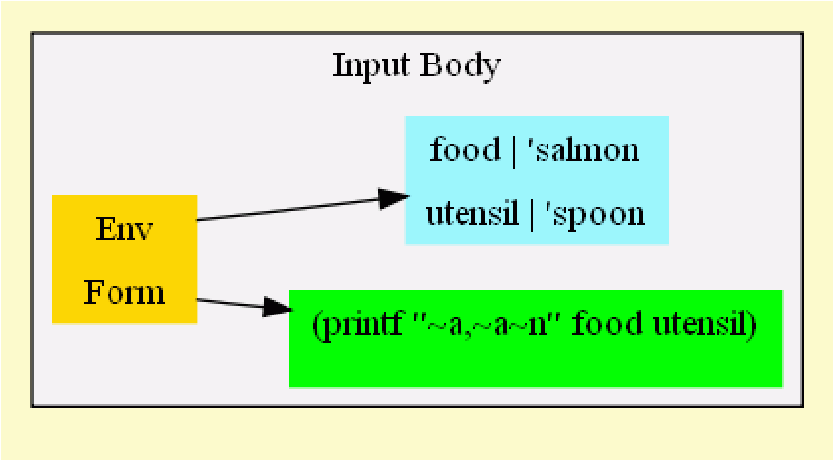
- Hygienic
  - Guarantee that expansion will not redefine existing name bindings
- Lexical Scope Twisting (Un-Hygienic)
  - By design allows you to modify existing bindings
  - Why? To introduce a **`return`** statement, or see "On Lisp" Anaphoric Macros
  - Anaphora: use of a grammatical substitute (as a pronoun or a pro-verb) to refer to the denotation of a preceding word or group of words.

Hygienic Macro 'hm' Template Source

```
[(_ body)
 #'(begin
     (define food 'perch)
     (define utensil 'fork)
     (printf "~a, ~a~n" food utensil)
     body)]
```

- `_` is the first argument of the pattern, and is always ignored. Using `_` is both loved and hated by Schemers
- **body** matches the entire form appearing as the argument to **hm**
- Everything following #' is the template

Hygienic Macro 'hm' Template Body




```
(let ([food 'salmon]
      [utensil 'spoon])
  (hm
   (printf
    "~a, ~a~n"
    food utensil)))
```

- Everything following **hm**, along with its environment, is the argument for **hm**

Hygienic Macro 'hm' Template Expansion

```
(let ([food 'salmon]
      [utensil 'spoon])
  (hm (printf "~a, ~a~n" food utensil)))
```

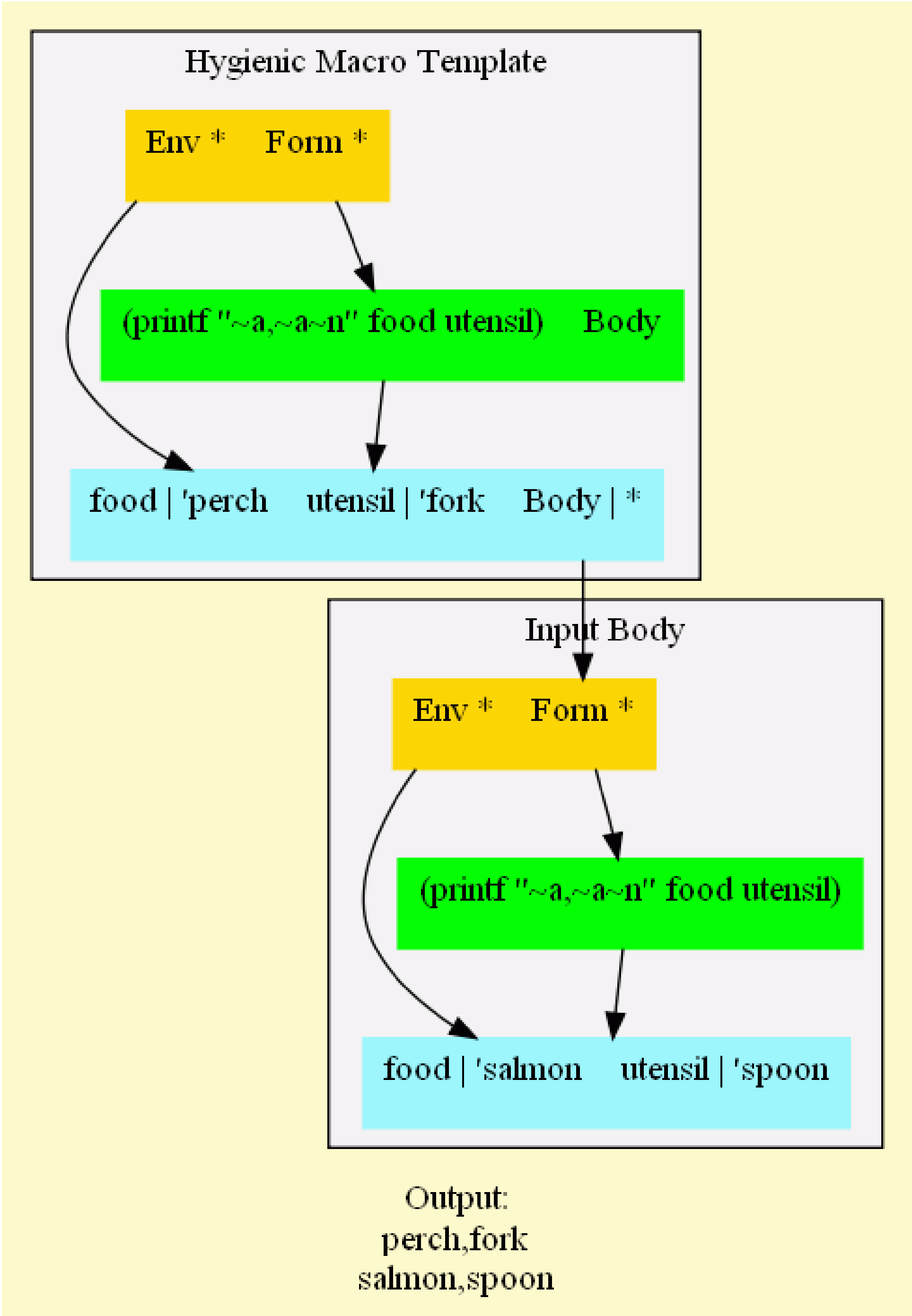
- Expands into

```
(let ([food 'salmon]
      [utensil 'spoon])
  (begin
    [define food 'perch]
    [define utensil 'fork]
    (printf "~a, ~a~n" food utensil)
    (printf "~a, ~a~n" food utensil)))
```

- On the next page is the interesting part; the **`printfs`** still use the correctly bound values

Hygienic Macro 'hm' Template Expansion

- [Review code in macro stepper and tracing arrows]
- Prints "perch, fork", then "salmon, spoon"


Hygienic Macro Template
Env *
Form *
(printf "~a,~a~n" food utensil)
Body
food | 'perch
utensil | 'fork
Body | *
Input Body
Env *
Form *
(printf "~a,~a~n" food utensil)
food | 'salmon
utensil | 'spoon
Output:
perch,fork
salmon,spoon

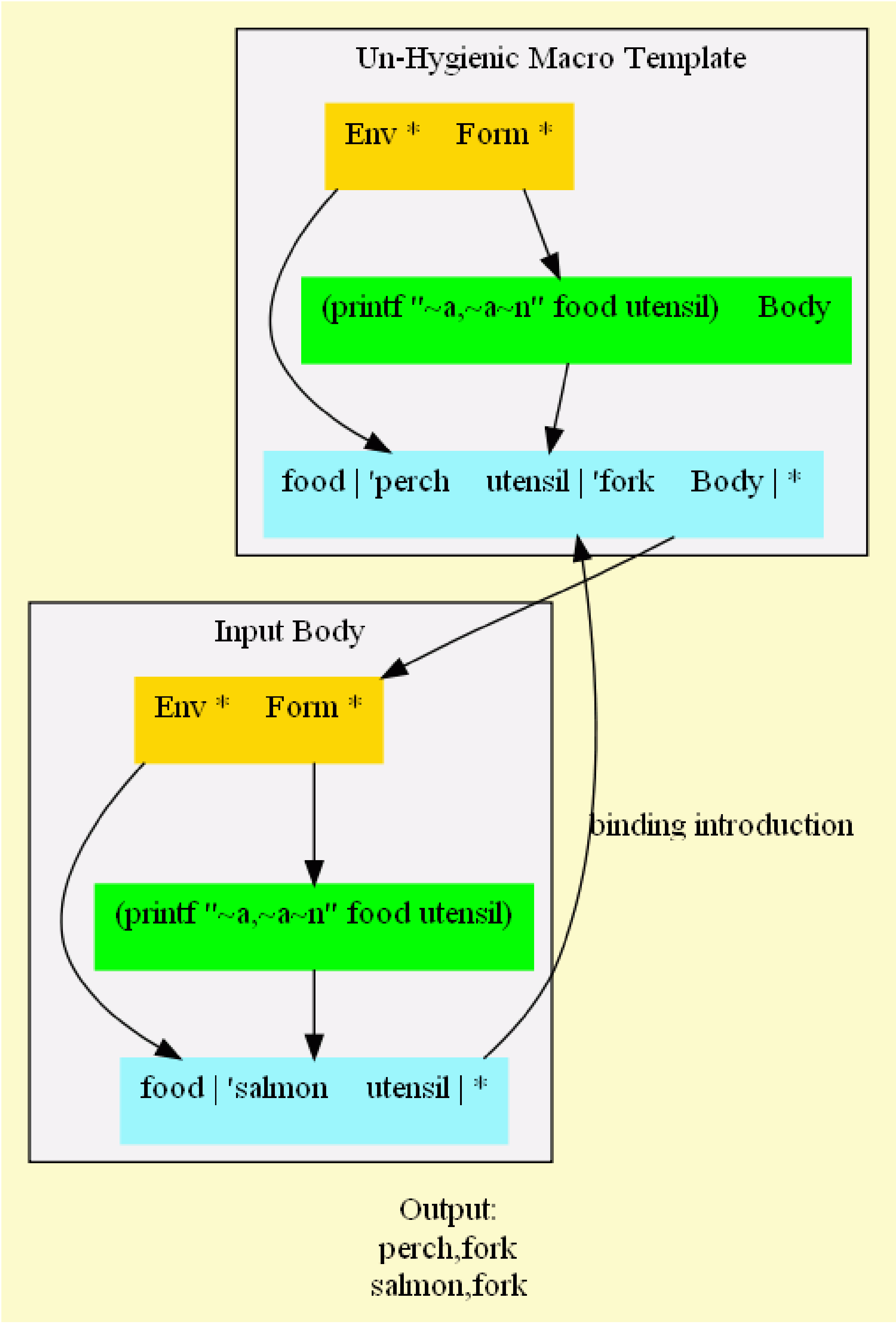
Un-Hygienic Macro 'uhm' Template Source

```
[(_ body)
 (with-syntax
     ([utensil
       (datum->syntax #'body 'utensil)])
   #'(begin
       (define food 'perch)
       (define utensil 'fork)
       (printf "~a, ~a~n" food utensil)
      body))]
```

- **`with-syntax`** provides the functionality to twist the lexical scope within the macro
- In this macro, **`utensil`** is inserted into the macro body's environment
- On the next page, you will see that the macro overrode the existing binding in the body

Un-Hygienic Macro 'uhm' Template Expansion

- [Review code in macro stepper and tracing arrows]
- Prints "perch, fork", then "salmon, fork"
- The template inserts a new binding into the body for **`utensil`**, breaking hygiene


Un-Hygienic Macro Template
Env * Form *
(printf "~a,~a~n" food utensil) Body
food | 'perch utensil | 'fork Body | *
Input Body
Env * Form *
(printf "~a,~a~n" food utensil)
food | 'salmon utensil | *
binding introduction
Output:
perch,fork
salmon,fork

Next Steps

- Implement a Primitive Object Syntax
- Implement Collision Detection
- Added Default Class Name & Methods Query

Prim Obj Stx: Pattern

```
(syntax-case stx (variables methods)
  [(define-object name
     (variables ([v-name v-val] ...))
     (methods ([m-name m-args m-body] ...)))])
```

- Reference: 4-prim-obj-stx.scm

Prim Obj Stx: Template

```
#'(define name
    (λ ()
      (define class-name (λ () 'name))
      (define method-names (λ () method-names-ls))
      (define method-names-ls
        '(class-name method-names m-name ...))
      (define v-name v-val) ...
      (define m-name (λ m-args m-body)) ...
      (λ (msg . args)
        (case msg
          [(class-name) (class-name)]
          [(method-names) (method-names)]
          [(m-name) (apply m-name args)] ...
          [else
           (raise 'err)]))))
```

Prim Obj Stx: Support Code

- invalid/duplicate identifier detection
  - Implementing using **`identifier?`** and **`bound-identifier=?`**

Thoughts

- Toys are for Learning
- The H-Word, and Other Hang-Ups
- Ideas Matter Most, Language Slavery, Innovation
- CoE: A Perfect "First Time"
- Thoughtful Teacher, Thoughtful Student
- As Difficult As [I] Make It
- The Midget vs. the Digits

Resources

- The Scheme Programming Language, Third Edition. R. Kent Dybvig
  - Inspiration for this task, the "K&R" book for Scheme
- PLT Scheme v4.02
  - mzscheme, DrScheme, Documentation, Discussion List
  - This presentation is written in Scheme, see bos-pres.scm and run.bat
  - Hit F5 to evalute and work with any code in the REPL
  - Use the #scheme module language. All code unit tested

Version

- $LastChangedDate: 2008-08-17 11:08:59 -0500 (Sun, 17 Aug 2008) $
- $LastChangedRevision: 2727 $
- $HeadURL: svn://osiris/scheme-bos-clug/tags/2.01/bos-pres.scm $