A Basic Object System Using Macros

A Talk With Grant Rettke

grettke@acm.org

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])
   (methods
    ([set-name (λ (arg) (set! name arg))]
     [get-name (λ () name)]
     [set-age-years (λ (arg) (set! age-years arg))]
     [get-age-years (λ () age-years)]
     [age-in-days (λ () (* age-years 365))]
     [typed-name (λ () (cons (get-name) (class-name)))
```

- Brackets may be used anywhere parentheses are used
  - Primarily to enhance readability
- (Getters and setters used to serve as the familiar; not Lisp style to do so)

Destination Code Test Sample

```
(test-case
 "define-primitive-object"
 (let ([obj (person)])
   (check-not-false obj "creation")
   (obj 'set-age-years 27)
   (check-eq?
    (obj 'get-age-years) 27 "get/set age-years")
   (obj 'set-name 'Joe)
   (check-eq? (obj 'get-name) 'Joe "get/set name")
   (check-eq?
    (obj 'age-in-days) (* 27 365) "age-in-days")
   (check-eq? (obj 'class-name) 'person "class-name")
   (check-eq?
    (length (obj 'method-names)) 10 "method-names")
   (check-eq?
    'person (cdr (obj 'typed-name)) "typed-name")))
```

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`**)
- This is how objects with lexical scope and encapsulation are created

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

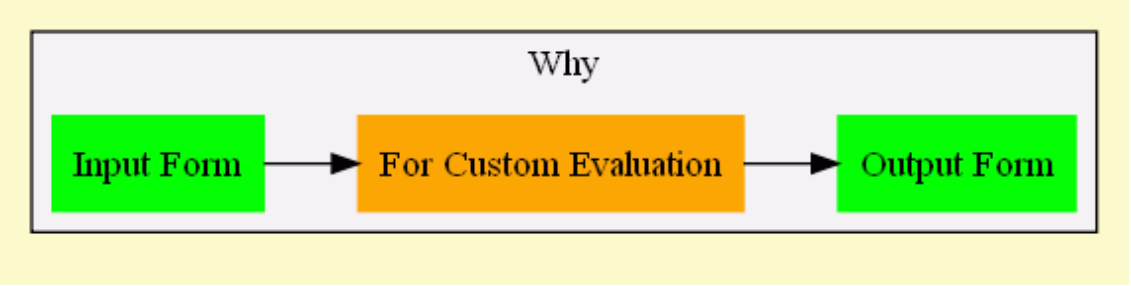
```
(define person
  (λ ()
    (define age-years #f)
    (define set-age-years
      (λ (arg) (set! age-years arg)))
    (define get-age-years (λ () age-years))
    (define age-in-days (λ () (* age-years 365)))
    (λ (msg . args)
      (case msg
        [(set-age-years) (apply set-age-years args)]
        [(get-age-years) (get-age-years)]
        [(age-in-days) (age-in-days)]
        [else (error "message not understood" msg)]))))
```

- The last λ expression is a closure which contains the lexical scope of the enclosing λ expression; this is how the "object" is created

Next Step: On To Generation

- High Level Macro Review
  - How much is worth discussing here?

Macros 1 - Why



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

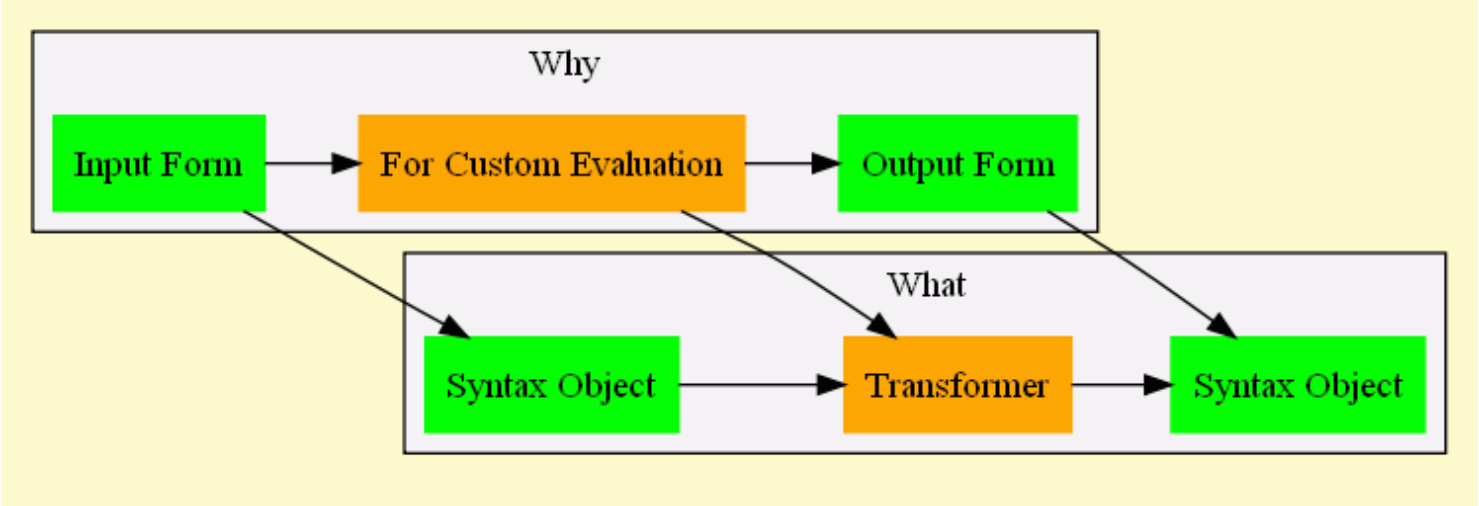
Macros 1 - Why - Example

- 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 x, it is null~n")
         (printf "x is ~a~n" (x))))
```

- **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**)

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

- 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 x, it is null~n")
       (printf "x is ~a~n" (x)))
```

- 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

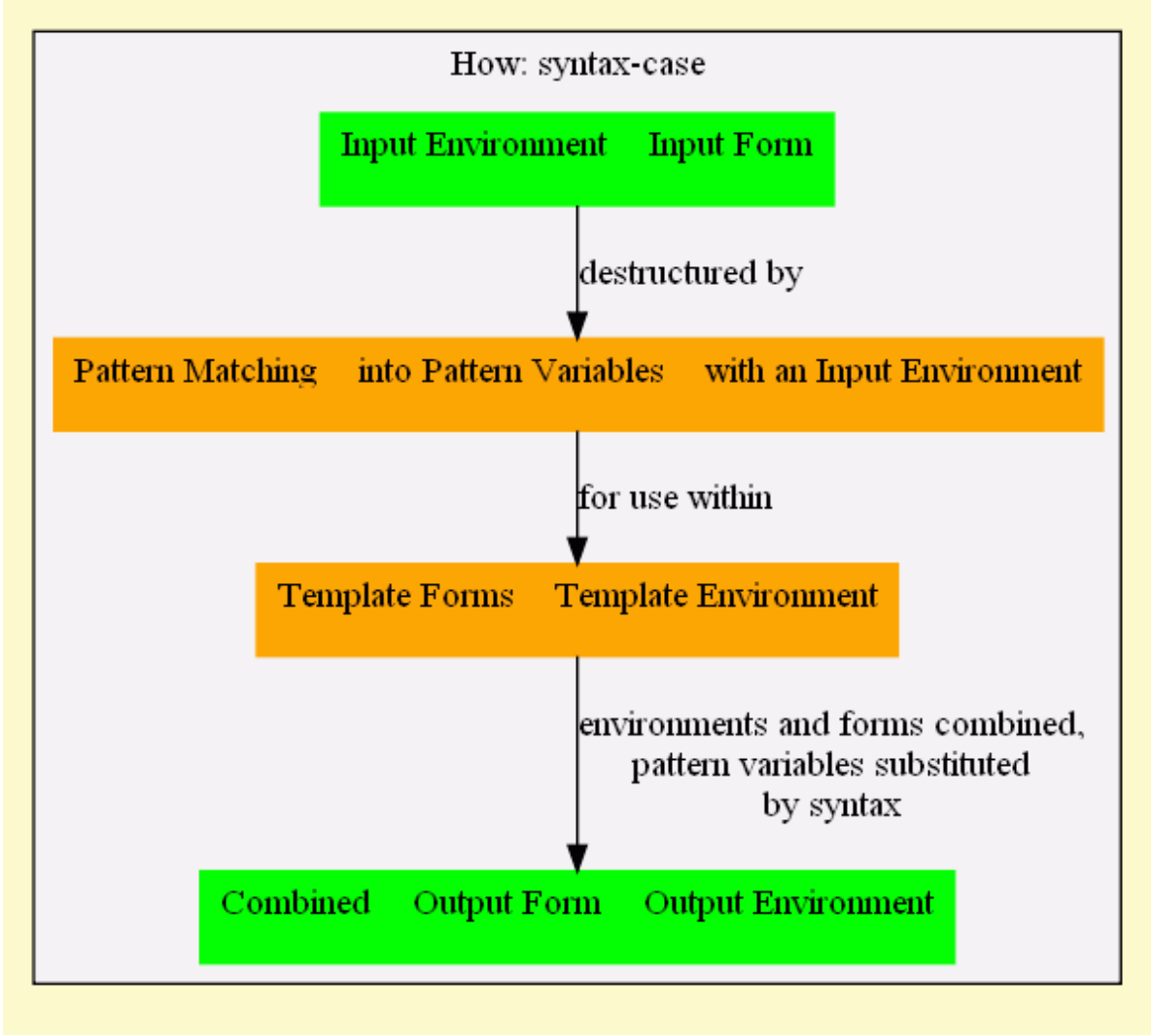
- Expands the template by replacing the pattern variables with their actual values and environment, and returns the resulting syntax-object

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

- Expanding into

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

Macros 3 - How - Visual



Two Kinds of Macros

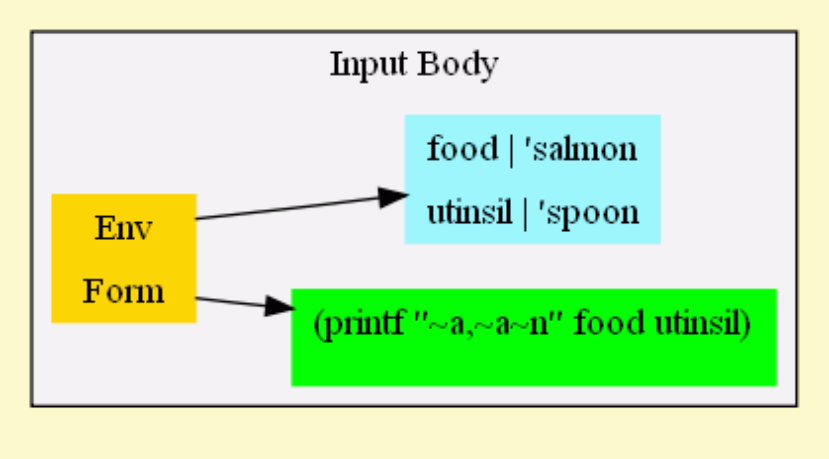
- Hygienic
  - Guarantee that expansion will not redefine existing name bindings
- Lexical Scope Twisting (Un-Hygienic)
  - By design allows you to modify existing bindings
  - Why? To introduce a `return` statement, or see "On Lisp" Anaphoric Macros

Hygienic Macro 'hm' Template Source

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

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

Hygienic Macro 'hm' Template Expansion

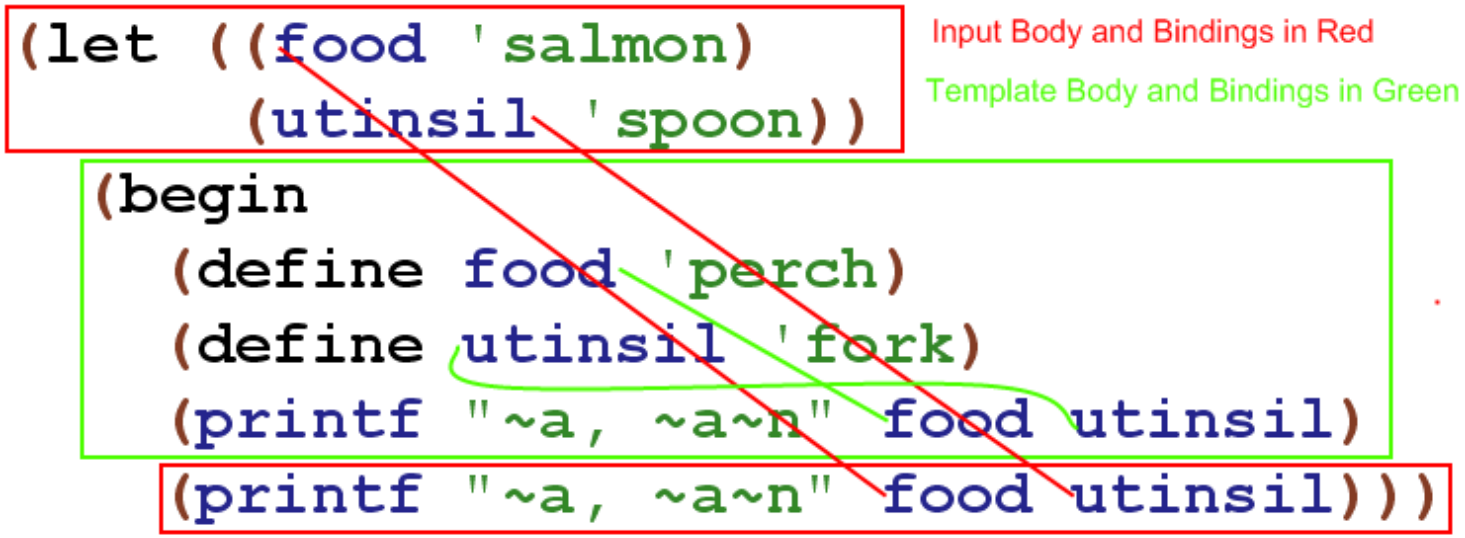
```
(let ([food 'salmon]
      [utinsil 'spoon])
  (hm (printf "~a, ~a~n" food utinsil)))
```

- Expands into

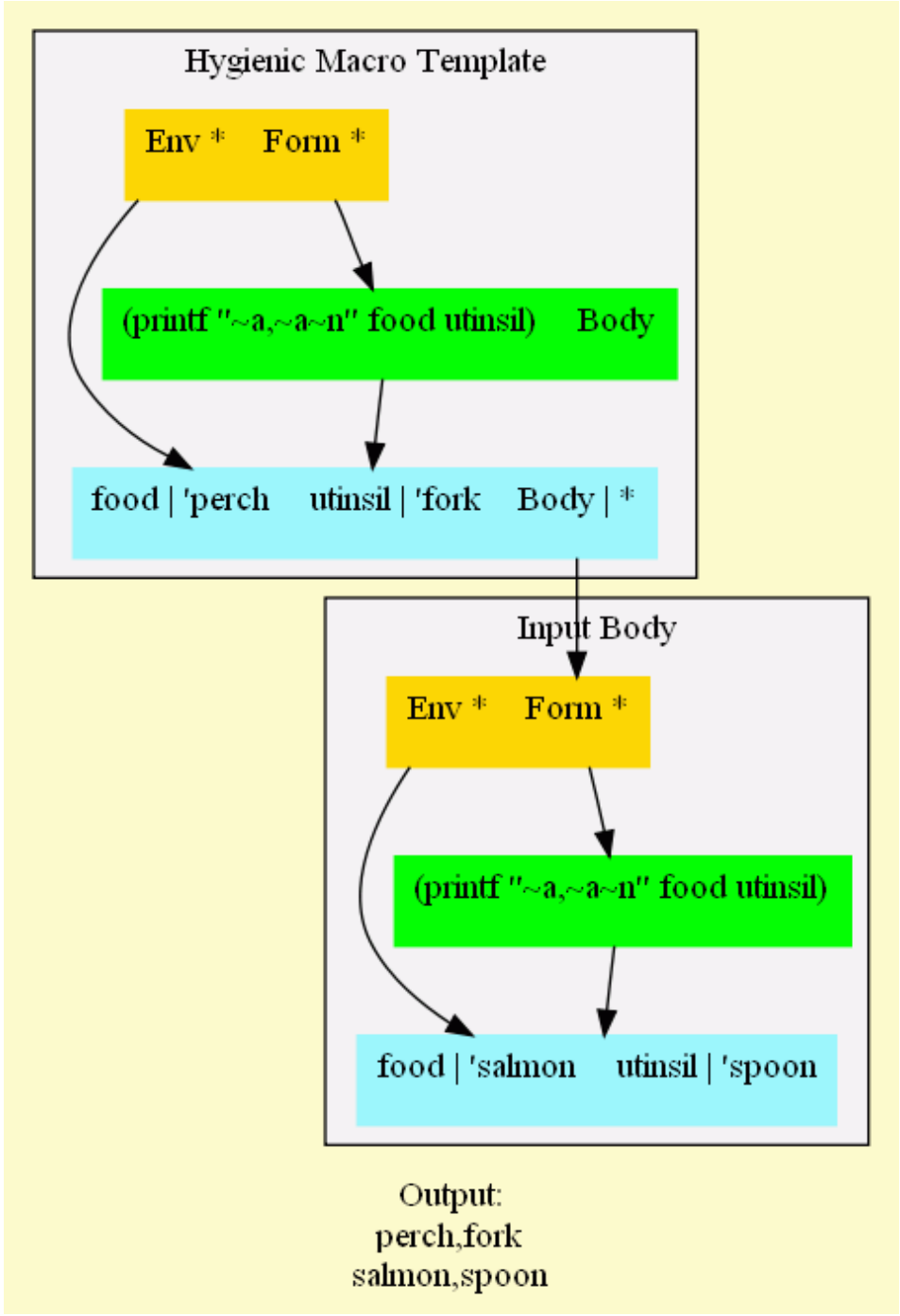
```
(let ([food 'salmon]
      [utinsil 'spoon])
  (begin
    [define food 'perch]
    [define utinsil 'fork]
    (printf "~a, ~a~n" food utinsil)
    (printf "~a, ~a~n" food utinsil)))
```

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

Hygienic Macro 'hm' Template Expansion



- Prints "perch, fork", then "salmon, spoon"

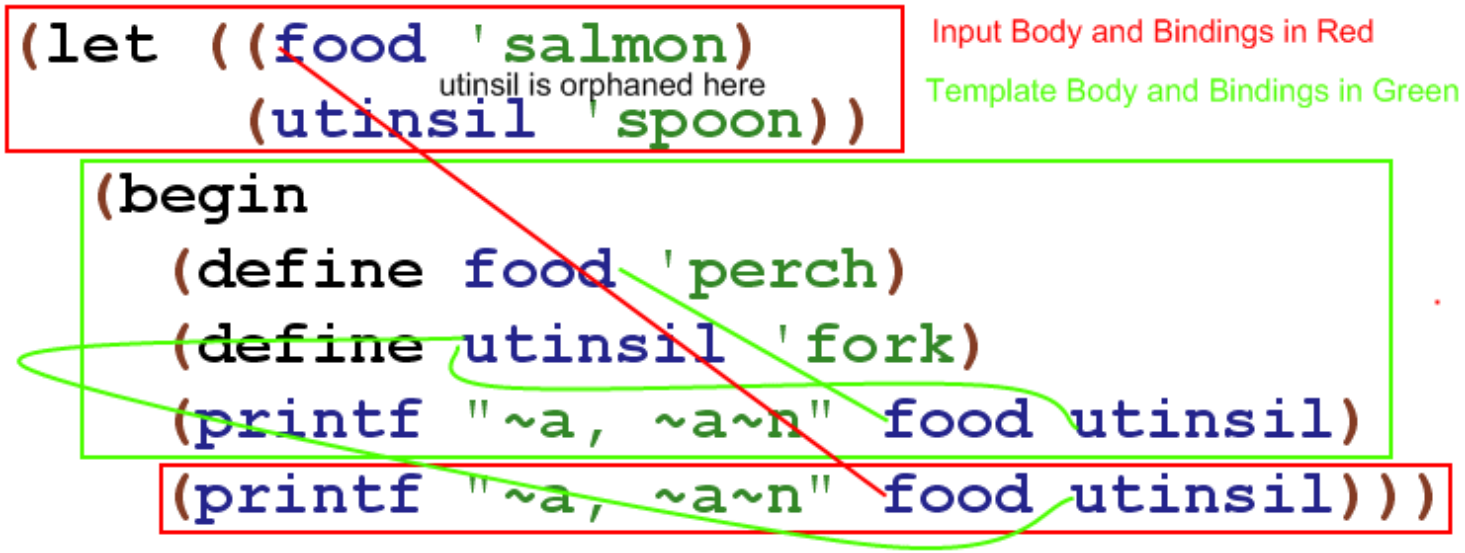

Hygienic Macro Template
Env * Form *
(printf "~a,~a~n" food utinsil) Body
food | 'perch utinsil | 'fork Body | *
Input Body
Env * Form *
(printf "~a,~a~n" food utinsil)
food | 'salmon utinsil | 'spoon
Output:
perch,fork
salmon,spoon

Un-Hygienic Macro 'uhm' Template Source

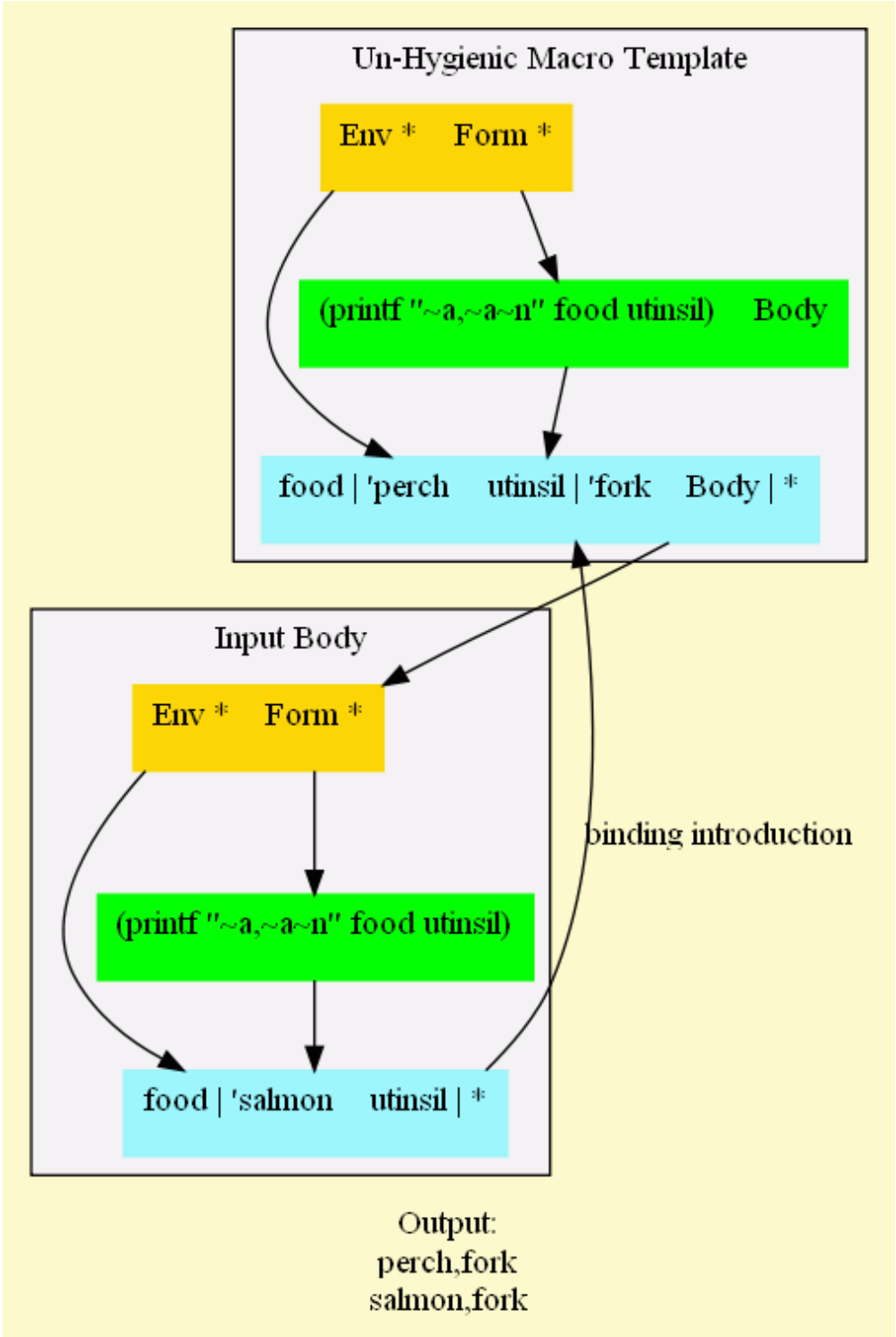
```
[(_ body)
 (with-syntax
     ([utinsil
       (datum->syntax-object #'body 'utinsil)])
   #'(begin
       (define food 'perch)
       (define utinsil 'fork)
       (printf "~a, ~a~n" food utinsil)
       body))]
```

- **`with-syntax`** provides the functionality to twist the lexical scope within the macro
- In this macro, **`utinsil`** 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



- Prints "perch, fork", then "salmon, fork"
- The template inserts a new binding into the body for **`utinsil`**, breaking hygiene


Un-Hygienic Macro Template
Env * Form *
(printf "~a,~a~n" food utinsil) Body
food | 'perch utinsil | 'fork Body | *
Input Body
Env * Form *
(printf "~a,~a~n" food utinsil)
food | 'salmon utinsil | *
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-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 m-body) ...
      (λ (msg . args)
        (case msg
          [(class-name) (class-name)]
          [(method-names) (method-names)]
          [(m-name) (apply m-name args)] ...
          [else
           (raise
            (string-append
             "message '"
             (symbol->string msg)
```

Prim Obj Stx: Support Code

- Macro Expander
  - View the expanded code
- with-syntax
  - Used to create unhygienic macros
- 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 v372
  - 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 module language. All code unit tested
- Web: Community-Scheme-Wiki, Schematics Scheme Cookbook
- [A history of] macro systems... http://lists.gnu.org /archive/html/chicken-users/2008-04/msg00013.html

Version

- $LastChangedDate: 2008-05-21 21:12:12 -0500 (Wed, 21 May 2008) $
- $LastChangedRevision: 1902 $
- $HeadURL: svn://osiris/scheme-bos-clug/tags/1.07/bos-pres.scm $