Hygienic Macros

Grant Rettke

grettke@acm.org

01: Values

01: Values

- 3 Tenets of Excellent Software

01: Values

- 3 Tenets of Excellent Software
  - Simple: It does a few things very, very well

01: Values

- 3 Tenets of Excellent Software
  - Simple: It does a few things very, very well
  - Highly Reliable: Correct, predictable, and stable

01: Values

- 3 Tenets of Excellent Software
  - Simple: It does a few things very, very well
  - Highly Reliable: Correct, predictable, and stable
  - Extensible: By the user

02: The Value of Macros

02: The Value of Macros

- LAMBDA: The Ultimate Imperative (1976)

02: The Value of Macros

- LAMBDA: The Ultimate Imperative (1976)
  - lambda, letrec, and if can be used to build nearly every construct

02: The Value of Macros

- LAMBDA: The Ultimate Imperative (1976)
  - lambda, letrec, and if can be used to build nearly every construct
  - Macros combined with call-by-value can build imperative from purely functional

02: The Value of Macros

- LAMBDA: The Ultimate Imperative (1976)
  - lambda, letrec, and if can be used to build nearly every construct
  - Macros combined with call-by-value can build imperative from purely functional
- LAMBDA: The Ultimate GOTO (1977)

02: The Value of Macros

- LAMBDA: The Ultimate Imperative (1976)
  - lambda, letrec, and if can be used to build nearly every construct
  - Macros combined with call-by-value can build imperative from purely functional
- LAMBDA: The Ultimate GOTO (1977)
  - A language should be so designed that one is encouraged to use a construct if, and only if, it is appropriate; it must also provide enough constructs to cover all reasonable programming constructs.

02: The Value of Macros

- LAMBDA: The Ultimate Imperative (1976)
  - lambda, letrec, and if can be used to build nearly every construct
  - Macros combined with call-by-value can build imperative from purely functional
- LAMBDA: The Ultimate GOTO (1977)
  - A language should be so designed that one is encouraged to use a construct if, and only if, it is appropriate; it must also provide enough constructs to cover all reasonable programming constructs.
  - A PL designer can not get it perfect for every case.

03: A syntax-rules

03: A syntax-rules

```
(from 1 upto 3
      (lambda (x) (display (format "~a~n" x))))
1
2
3
```

03: A syntax-rules

```
(from 1 upto 3
      (lambda (x) (display (format "~a~n" x))))
1
2
3

    (define-syntax from
      (syntax-rules (upto)
        ([from x upto y fn]
         [let ([finish (+ y 1)]
               [add1 (lambda (n) (+ n 1))])
           (let loop ([cur x])
             (if (not (= cur finish))
                 (begin
                   (fn cur)
                   (loop (add1 cur))))))])))
```

04: Scoping Concerns

```
(define-syntax from
  (syntax-rules (upto)
    ([from x upto y fn]
     [let ([finish (+ y 1)]
           [add1 (lambda (n) (+ n 1))])
       (let loop ([cur x])
         (if (not (= cur finish))
             (begin
               (fn cur)
               (loop (add1 cur)))))]))))
```

04: Scoping Concerns

```
(define-syntax from
  (syntax-rules (upto)
    ([from x upto y fn]
     [let ([finish (+ y 1)]
           [add1 (lambda (n) (+ n 1))])
       (let loop ([cur x])
         (if (not (= cur finish))
             (begin
               (fn cur)
               (loop (add1 cur)))))]))))
```

- Integrity of template bindings

04: Scoping Concerns

```
(define-syntax from
  (syntax-rules (upto)
    ([from x upto y fn]
     [let ([finish (+ y 1)]
           [add1 (lambda (n) (+ n 1))])
       (let loop ([cur x])
         (if (not (= cur finish))
             (begin
               (fn cur)
               (loop (add1 cur)))))]))))
```

- Integrity of template bindings
- Integrity of (pattern) input bindings

04: Scoping Concerns

```
(define-syntax from
  (syntax-rules (upto)
    ([from x upto y fn]
     [let ([finish (+ y 1)]
           [add1 (lambda (n) (+ n 1))])
       (let loop ([cur x])
         (if (not (= cur finish))
             (begin
               (fn cur)
               (loop (add1 cur)))))]))))
```

- Integrity of template bindings
- Integrity of (pattern) input bindings
- "Hygiene" maintains lexical scoping for macros

05: Dynamic Scoping with Elisp. Do you really like it?

```
(defvar x 42)
(defun sample ()
  (message "%s" x))
(defun dynamic-scope-sample ()
  (let ((x 666))
    (sample)))
(sample)
"42"
(dynamic-scope-sample)
"666"
```

06: Dynamic Scoping in the Small

06: Dynamic Scoping in the Small

- Manageable in simple cases

06: Dynamic Scoping in the Small

- Manageable in simple cases
  - ... but do we only care about simple cases?

07: Dynamic Scoping in the Large

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large
- Can we manage scoping issues past 2 steps?

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large
- Can we manage scoping issues past 2 steps?
  - 5?

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large
- Can we manage scoping issues past 2 steps?
  - 5?
  - 10?

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large
- Can we manage scoping issues past 2 steps?
  - 5?
  - 10?
  - 20?

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large
- Can we manage scoping issues past 2 steps?
  - 5?
  - 10?
  - 20?
  - 50?

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large
- Can we manage scoping issues past 2 steps?
  - 5?
  - 10?
  - 20?
  - 50?
  - 100?

07: Dynamic Scoping in the Large

- Our goal is to support programming in the large
- Can we manage scoping issues past 2 steps?
  - 5?
  - 10?
  - 20?
  - 50?
  - 100?
- If we want macros to be first class citizens in large scale development, hygiene seems more likely to succeed (computer vs. human)

08.1: syntax-case

08.1: syntax-case

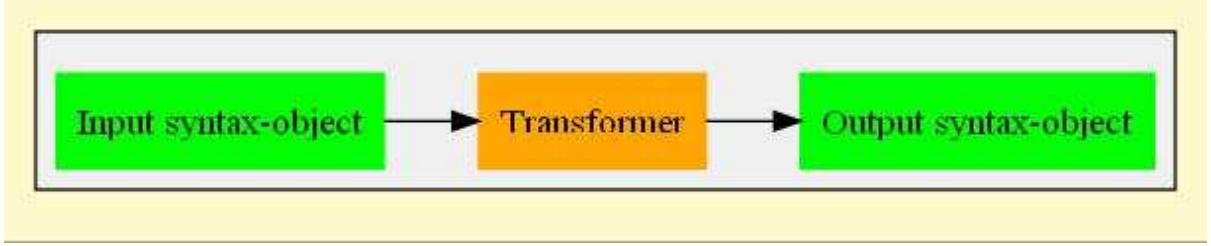
- Anaphoric Macros: being a word or phrase that takes its reference from another word or phrase and especially from a preceding word or phrase

08.1: syntax-case

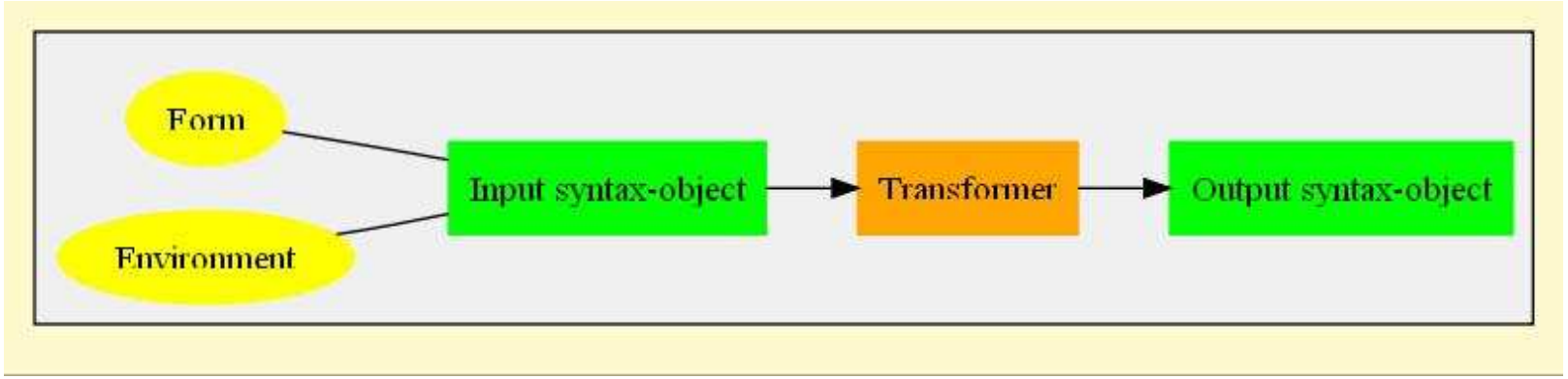
- Anaphoric Macros: being a word or phrase that takes its reference from another word or phrase and especially from a preceding word or phrase

```
(aif (+ 10 10) it -1)
```

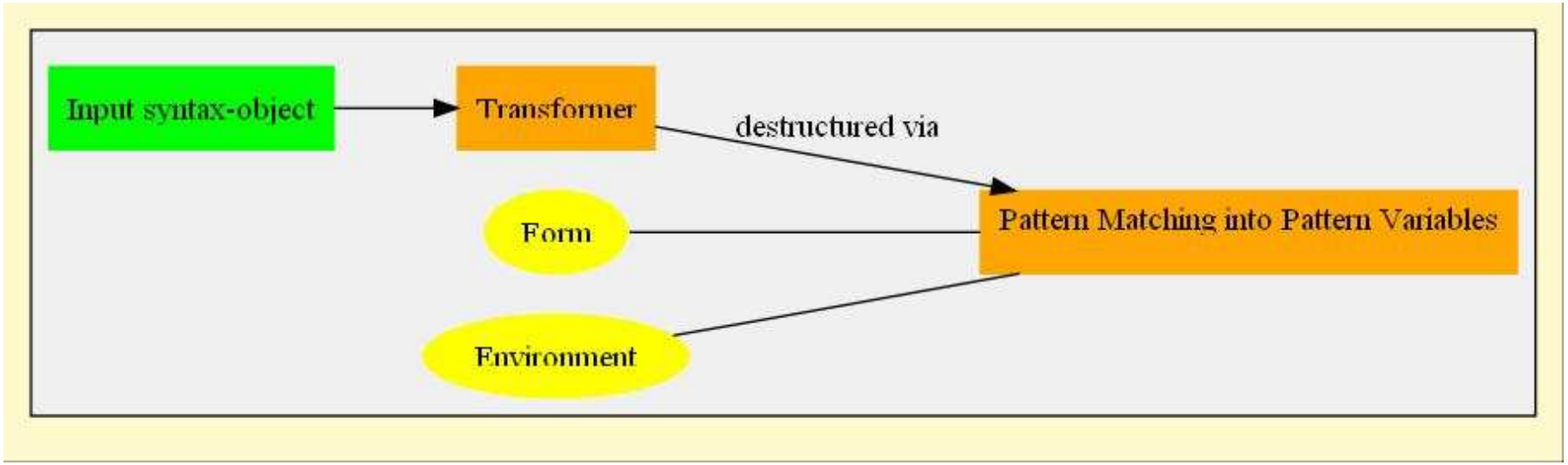
08.2: syntax-case



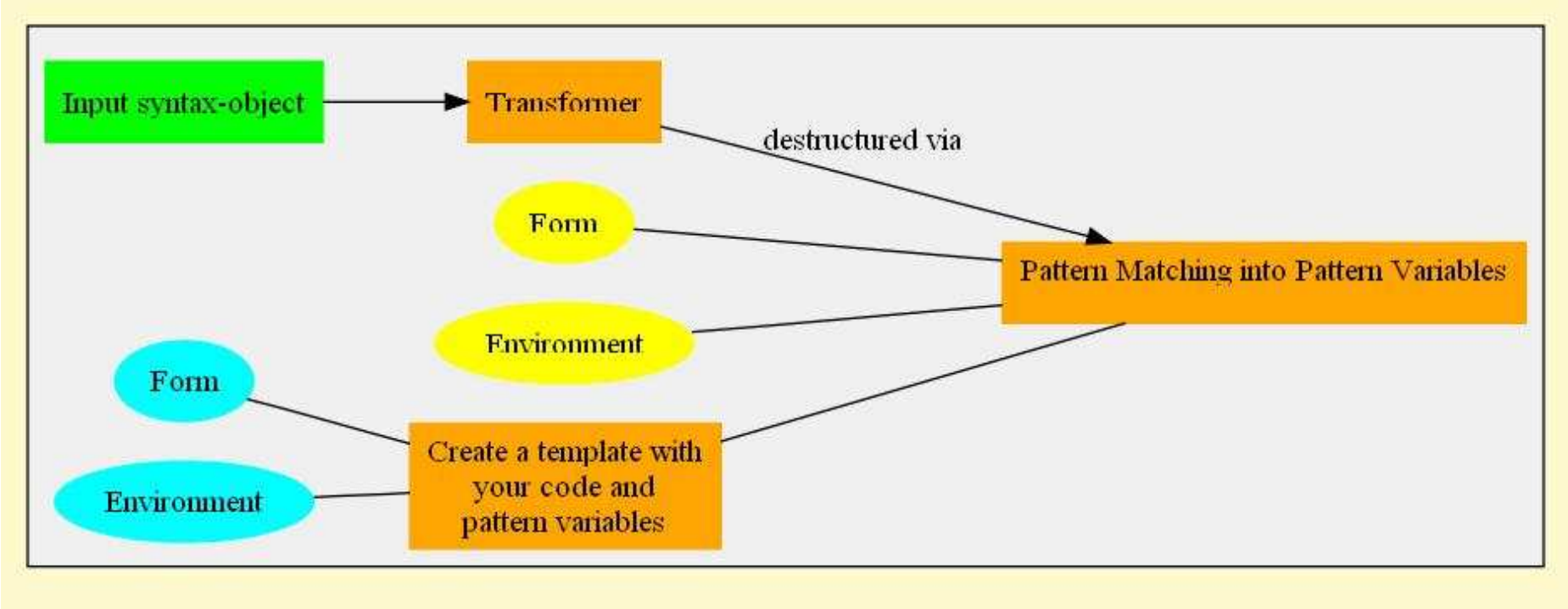
08.3: syntax-case



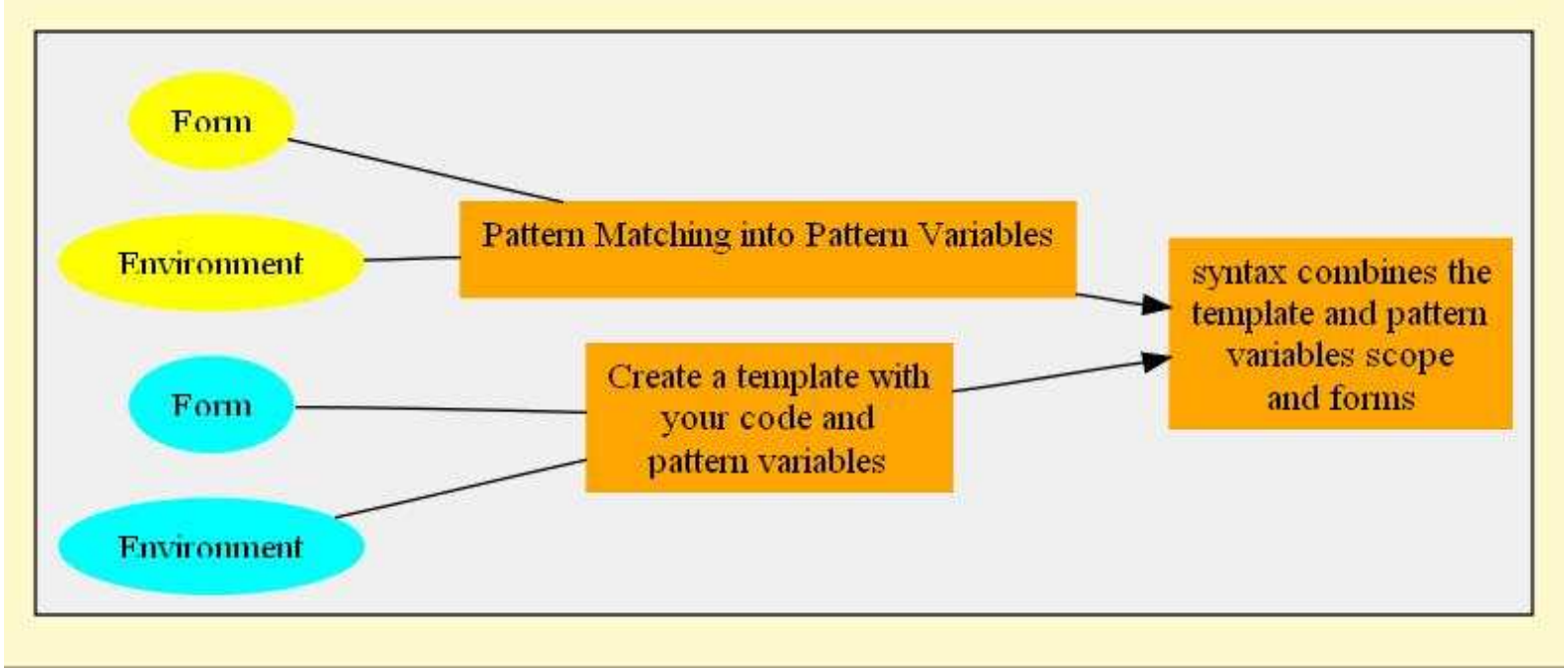
08.4: syntax-case



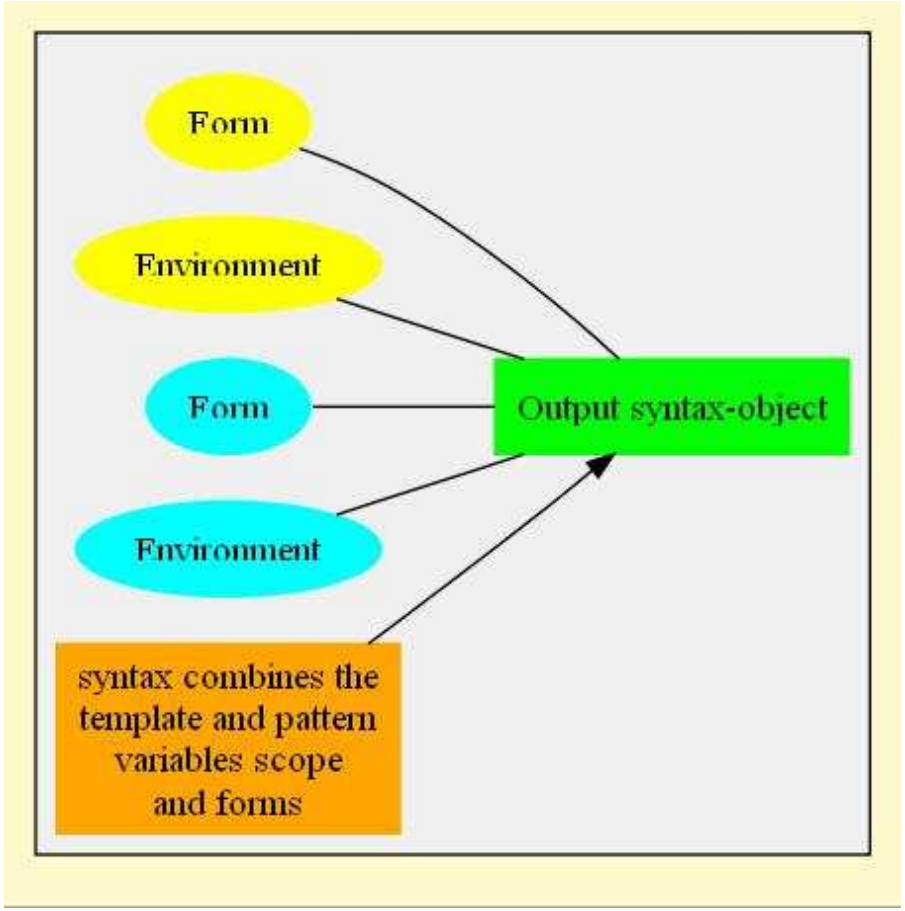
08.5: syntax-case



08.6: syntax-case



08.7: syntax-case



08.8: syntax-case

08.8: syntax-case

```
(define-syntax aif
  (lambda (stx)
    (syntax-case stx ()
      ([aif test-form then-form else-form]
       [with-syntax ([it (datum->syntax #'aif 'it)])
         #'(let ((it test-form))
             (if it
                 then-form
                 else-form))]))))
```

09: Experience

09: Experience

- lambda and syntax-rules do just fine

09: Experience

- lambda and syntax-rules do just fine
- Many folks prefer defmacro

09: Experience

- lambda and syntax-rules do just fine
- Many folks prefer defmacro
- syntax-case is a superset of syntax-rules; can do anything defmacro allows

09: Experience

- lambda and syntax-rules do just fine
- Many folks prefer defmacro
- syntax-case is a superset of syntax-rules; can do anything defmacro allows
- Respect macros; it is easy to get burned even with hygiene

10: Fun

10: Fun

```
(define-syntax sequencing
  (syntax-rules ()
    [(_ expression) expression]
    [(_ expression expressions ...)
     ((lambda (ignored)
        (sequencing expressions ...))
      expression)]))
```