

CSC324H Principle of Programming Languages (Week 7)

Yilan Gu

yilan@cdf.toronto.edu

<http://www.cs.toronto.edu/~yilan/324f09>

Introduction

1

Where are we now?

- ✓ Introduction
- ✓ Formal language specification
- Scheme
 - ✓ Basic types, functions
 - ✓ Different recursions, efficiency and tail recursion
 - ✓ Let, let*, letrec
 - ✓ Unbounded lambda
 - ✓ Higher-order functions
 - ✓ Strings, vectors, other useful procedures
 - ✓ Functional programming (FP) and formal proofs
 - **Macro (LAST LECTURE OF SCHEME)***
- ✓ Theory – lambda calculus

* partially based on G. Baumgartner's lecture notes

2

Preliminaries

- **Macros (syntactic extensions)**
 - simplify and regularize repeated patterns in a program
 - introduce syntactic forms with new evaluation rules
 - perform transformations to help make programs more efficient (e.g., [automata via macros](#))
- Several different ways of defining macros (Mitchell, Chapter 8), and we will introduce the basic one
- Format:
 - syntax: (**define-syntax** keyword transformer)
 - returns: unspecified
- Transformer with **syntax-rules**
 - syntax: (syntax-rules (identifier ...) (pattern template) ...)
 - returns: a transformer

3

4

Format of Pattern/Template

- a (pattern) variable (e.g., `_`)
- an identifier
- a list of the form $(P_1 \dots P_n)$
- an improper list of the form $(P_1 \dots P_n . P_x)$
- a list of the form $(P_1 \dots P_k P_e \textit{ellipsis} P_{m+1} \dots P_n)$, where *ellipsis* is the identifier `...`
- an improper list of the form $(P_1 \dots P_k P_e \textit{ellipsis} P_{m+1} \dots P_n . P_x)$, where *ellipsis* is the identifier `...`
- a pattern datum (any nonlist, nonvector, nonsymbol object)
- etc. (such as vectors, for complete list of patterns, see Chapter 8.2)

5

Some Simple Examples

```
(define-syntax let*  
  (syntax-rules ()  
    ((_ () e1 e2 ...) (let () e1 e2 ...))  
    ((_ ((i1 v1) (i2 v2) ...) e1 e2 ...) (let ((i1 v1))  
      (let* ((i2 v2) ...) e1 e2 ...)))  
  )  
)
```

```
(define-syntax cond  
  (syntax-rules (else)  
    ((_ (else e1 e2 ...) (begin e1 e2 ...))  
      ((_ (e0 e1 e2 ...) (if e0 (begin e1 e2 ...)))  
        ((_ (e0 e1 e2 ...) c1 c2 ...) (if e0 (begin e1 e2 ...) (cond c1 c2 ...))))))
```

6

Some Simple Examples

```
(define-syntax d1  
  (syntax-rules ()  
    ((_ (fun ... . rest) pgm ...) (display (quote rest)))  
  ))
```

```
(define-syntax d2  
  (syntax-rules ()  
    ((_ (fun . rest) pgm ...) (display (quote rest)))  
  ))
```

7

A Special Usage of ...

- (... **template**)
 - lose its special meaning
 - is identical to **template**
 - allows syntactic extensions to expand into syntax definitions containing ellipses.

```
(define-syntax be-like-begin  
  (syntax-rules ()  
    ( ( _ name) ; pattern  
      (define-syntax name ; starting of template  
        (syntax-rules ()  
          ((_ e0 e1 (... ...))  
            (begin e0 e1 (... ...))))))
```

8

A Special Usage of ...

```
> (be-like-begin sequence) ; evaluates to
(define-syntax sequence
  (syntax-rules ()
    ( ( _ e0 e1 ...)
      (begin e0 e1 ...))))
```

```
> (sequence
  (display "Hi mom")
  (newline))
```

Hi mom

Some Characters of Macros in Scheme

9

10

Recall: macros in C/C++

- User-defined syntactic forms are also called **macros**.
- Let's review macros in C/C++. They are essentially textual substitutions.

```
#define SQUARE(x) x*x
```

- This has a problem:
4 * SQUARE(2 + 3) gets expanded to 4 * 2 + 3 * 2 + 3, completely messing up the intended meaning.
- To fix SQUARE:

```
#define SQUARE(x) ((x)*(x))
```

Doesn't come up in Scheme, since all expressions are fully parenthesized.

11

Another Example

Consider now a macro to swap the values of two variables:

```
#define SWAP(v1,v2) int t = v1; v1 = v2; v2 = t;
```

This has a problem if t is one of the variables we want to swap:

```
SWAP(t,v): int t = t; t = v; v = t;
```

==> **The hygiene problem**

In Scheme – **no problem**:

```
(define-syntax swap
  (syntax-rules ()
    ((swap v1 v2) ; names in code matched to v1 and v2 refer to the caller's names
      (let ((t v1)) ; other names in result template (let, t, set!) refer to names where
        (set! v1 v2) ; macro is defined
        (set! v2 t))))
```

12

Hygienic Transformation

- The processor responsible for transforming the patterns of the input form into an output form detects symbol clashes and resolves them by temporarily changing the names of symbols
- We get much of the usual effect of procedures, with the ability to pass code that doesn't lose its meaning:
 - meaning of non-parameters comes from environment where swap is defined
 - meaning of parameter code (if evaluated) comes from caller's environment

13

Hygienic Transformation

- Here is an annotated expansion of (swap t v):
(swap-let ((swap-t caller-t))
 (swap-set! caller-t caller-v)
 (swap-set! caller-v swap-t))
- If we haven't redefined let and set! where swap was made or called, we can just think of it as:
(let ((swap-t caller-t))
 (set! caller-t caller-v)
 (set! caller-v swap-t))
- And if we think of it as being expanded where (swap t v) is called, it's equivalent to:
(let ((swap-t t))
 (set! t v)
 (set! v swap-t))

14

Applicative vs Normal Order

- Code is a tree, and is evaluated by a mixture of top-down/outside-in (**Normal order**) and bottom-up/inside-out (**Applicative order**) processing.
- Call-by-value is bottom-up: the procedure executes only after the arguments have been evaluated.
- Syntactic-forms are top-down: they control the processing of their arguments. For example:

```
(define-syntax s
  (syntax-rules ()
    ((s (x y) y)))
```

(s (not #t)) ;transforms to code #t, when run evaluates to #t

(s (s (not #t))) ;transforms to code (not #t)
;when run evaluates to #f

'(s (s (not #t))) ;shorthand for (quote (s (s (not #t))))

15

The Power of Macros

16

Simplifying Syntax: Named-Let

- named-let

```
(define (sum-of-squares n)
  (let s-o-s ((s 0) (i 1))
    ; s-o-s procedure of two arguments s and i
    ; called initially with 0 and 1
    (if (> i n) s
        (s-o-s (+ s (* i i)) (+ i 1))))))
```
- This is short for:

```
(define (sum-of-squares n)
  (letrec ((s-o-s
            (lambda (s i)
              (if (> i n) s (s-o-s (+ s (* i i)) (+ i 1))))))
    (s-o-s 0 1)))
```

17

Simplifying Syntax: Named-Let

- define named-let ourselves:

```
(define-syntax named-let
  (syntax-rules ()
    ((named-let <name> ((<var> <init>) ...) <body> ...)
     (letrec ((<name> (lambda (<var> ...)
                      ; puts one of each <var> here
                      <body> ...)))
       (<name> <init> ...)))) ; puts one of each <init> here
```
- Note:
 - “<” and “>” have no special meanings, just a convention for non-terminal
 - the variables and initial values are together, but the ... can loop over them individually.

18

Define New/Redefine Evaluation Rules

```
> (let ((x #t) (y #f))
  (define-syntax and
    (syntax-rules ()
      ((_) #f)
      ((_ e) e)
      ((_ e1 e2 e3 ...)
       (let ((t e1)) (if t (and e2 e3 ...))))))
  (and x y))
#t

> ((let ((x #t) (y #f)) (and x y))
  #f
```

19

An Example of using Helper Macros

```
(define-syntax rec
  (syntax-rules ()
    ((_ x e) (letrec ((x e)) x))))

(map (rec sum
  (lambda (x)
    (if (= x 0)
        0
        (+ x (sum (- x 1))))))
  '(0 1 2 3 4 5))

=> (0 1 3 6 10 15)
```

20

An Example of using Helper Macros

```
(define-syntax rec
  (syntax-rules ()
    ((_ x e) (letrec ((x e)) x))))
```

```
(map (rec sum
      (lambda (x)
        (if (= x 0)
            0
            (+ x (sum (- x 1))))))
     '(0 1 2 3 4 5))
```

=> (0 1 3 6 10 15)

An Example of using Helper Macros

```
(define-syntax rec
  (syntax-rules ()
    ((_ x e)
     (letrec ((x e)) x))))
```

- Using rec, we can define the full let (both unnamed and named) as follows.

```
(define-syntax let
  (syntax-rules ()
    ((_ ((x v) ...) e1 e2 ...)
     ((lambda (x ...) e1 e2 ...) v ...))
    ((_ f ((x v) ...) e1 e2 ...)
     ((rec f (lambda (x ...) e1 e2 ...) v ...))))
```

21

22

An Example of using Helper Macros

- It is the same as define let directly using letrec:

```
(define-syntax let
  (syntax-rules ()
    ((_ ((x v) ...) e1 e2 ...)
     ((lambda (x ...) e1 e2 ...) v ...))
    ((_ f ((x v) ...) e1 e2 ...)
     ((letrec ((f (lambda (x ...) e1 e2 ...)))
        f v ...))))
```

23

Readings: Dybvig, Chapter 8.1,8.2

24