Programming Languages (Langages Evolué)

Roel Wuyts

Scheme

Outline

- Variables and Scoping
- Macroes
- Imperative Programming
- Modeling Objects
- Streams
- Conclusion

Getting in the mood again :-)

```
;; something with lists
(define (enumerate-interval low high)
  (if (> low high)
      '()
      (cons low (enumerate-interval (+ low 1) high))))
(enumerate-interval 10 16) => (10 11 12 13 14 15 16)


;;a higher-order procedure
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
(accumulate + 0 '(1 5 9 14)) => 29
```

Variables and Scoping

- Example

```
n => ERROR:  Undefined global variable n
(define n 5)
n => 5
(* 4 n) => 20
```

- Variables and scoping:
  - (define name expression)
  - (lambda...
  - (let...

Local Variables, lambda

```
(define (f x)
      (/ (+ (square x) 1)
         (- (square x) 1)))
(define (f x)
      (define y (square x))
      (/ (+ y 1) (- y 1)))
y => error


(define x 20)
(define add2
    (lambda (x)
       (set! x (+ x 2))
   x))

(add2 3) => 5
(add2 x) => 22
x => 20
```

Local Variables and let

```
(define (f x)
      (define (f-help y)
        (/ (+ y 1) (- y 1)))
      (f-help (square x)))


(define (f x)
      ((lambda (y)
         (/ (+ y 1) (- y 1)))
       (square x)))


 (define (f x)
      (let (   (y (square x))   )
         (/ (+ y 1) (- y 1))))
```

Let & Lambda

- (let ((*variable-1 expression-1*)
  (*variable-2 expression-2*)
  . . .
  (*variable-n expression-n*))
  body)

  is syntactic sugar for

  ((lambda (*variable-1* ... *variable-n*) *body*)
  expression-1
  . . .
  expression-n)

Let and Lambda: Example

```
(let ((x (* 4 5))
       (y 3))
  (+ x y))

((lambda (x y) (+ x y))
 (* 4 5) 3)
```

Let examples

```
(let ( (x 2)
       (y 3) )
  (* x y))                                        =>  6

(let ( (x 2)
       (y 3))
  (let ( (x 7)
         (z (+ x y)) )
    (* z x)))                                     =>  35
```

More *let* Examples

```
(let ( (x 1)
       (y 2)
       (z 3))
  (list x y z))          => (123)


(define x 20)
(let ( (x 1)
       (y x))
  (+ x y))               => 21


(let ( (cons (lambda (x y) (+ x y))) )
  (cons 1 2))      => 3
```

Let*

```
(let ( (x 2)
       (y 3))
  (let* ( (x 7)
          (z (+ x y)))
    (+ z x)))                     =>  17

(let* ( (x 1)
        (y x))
  (+ x y))                       => 2
;; equivalent to
(let ( (x 1) )
  (let ( (y x) )
    (+ x y)))                   => 2
```

Letrec

- local-even? and local-odd? in the initializations don't refer to the lexical variables themselves.

```
(let ( (local-even?
        (lambda (n) ( if (= n 0) #t (local-odd? (- n 1)))))
       (local-odd?
        (lambda (n) ( if (= n 0) #f (local-even? (- n 1))))))
  (list (local-even? 23) (local-odd? 23)))
```

- Changing the let to a let* won't work
  - the local-odd? in local-even? 's body still points elsewhere.

Letrec Examples

```
(letrec ( (local-even?
              (lambda (n) ( if (= n 0) #t (local-odd? (- n 1)))))
          (local-odd?
              (lambda (n) ( if (= n 0) #f (local-even? (- n 1))))))
  (list (local-even? 23) (local-odd? 23)))


(letrec ((countdown (lambda (i)
                       (if (= i 0) 'liftoff
                           (begin
                             (display i)
                             (newline)
                             (countdown (- i 1)))))))
  (countdown 10))
```

Named let

- Previous example can be written shorter using a *named let*
- Example

```
(let countdown ((i 10))
  (if (= i 0) 'liftoff
      (begin
        (display i)
        (newline)
        (countdown (- i 1)))))
```

Macroes

- Used to define special forms
- Specifies a purely textual transformation from code text to other code text.
- Example

```
(define-macro when
  (lambda (test . branch)
    (list 'if test
      (cons 'begin branch))))

(when (< x 2)
   (display "something")
   (display "something else))
```

Rewriting

- Takes an S-expression, produces an S-expression

```
(when (< x 2)
   (display “something”)
   (display “else”))
```

```
(apply
   (lambda (test . branch)
      (list ‘if test
        (cons ‘begin branch)))
   ‘((< x 2)
      (display “something”)
      (display “else”)))
```

```
(if (< x 2)
   (begin
     (display “something”)
     (display “else”))
```

Using templates

- Return of the backquote
- Defining *when* again:
```
(define-macro when
  (lambda (test . branch)
    `(if ,test
      (begin ,@branch))
```
- Inserting arguments in backquoted expressions:
  - , insert result of evaluating expression
  - ,@ removes outermost parentheses, then evaluates and inserts the result

Evaluation of arguments

- We already saw that this does not work:

```
(define (my-if test true false)
   (cond (test true)
   (else false)))

(my-if (> 3 2) (display "t") (display "n"))
   => tn
```

- Using a macro:

```
(define-macro my-if
   (test true false)
   `(cond (,test ,true)
          (else ,false)))

(my-if (> 3 2) (display "t") (display "n"))
   => t
```

Imperative Programming

- By default Scheme has no side effects
  - No variables
- Imperative programming: manipulate variables
  - Destructively change contents in memory
- Three special forms used:
  - set!
  - set-car!
  - set-cdr!

set!

- (set! *name new-value*)
- Example

```
(define x 5)
(+ 1 x) => 6

(set! x 10)
x => 10
```

Bank account example

```scheme
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
    (begin
      (set! balance (- balance amount))
      balance)
    "insufficient funds"))

(withdraw 20) => 80
(withdraw 10) => 70
```

Bank account with local state

```
(define withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
      "insufficient funds"))))

(withdraw 20) => 80
(withdraw 10) => 70
```

Withdraw processor

```
(define (make-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))

(define my-withdrawer (make-withdraw 100))
(my-withdrawer 20) => 80
(my-withdrawer 10) => 70
```

Cost of Assignment

- Hard to model objects and assignment
- Without assignment, two evaluations produce same result
  - compute mathematical functions
- When are things "the same"?
- Order of assignments is important
  - Amazing that we can do it actually!
- Concurreny is really problematic

Modeling Objects

- Classes are created as dispatch tables
- Example

```
(define (number-class x)
   (lambda (message)
      (cond
         ((eq? message 'show) (display x))
         ((eq? message 'inc) (set! x (+ x 1)))
         ((eq? message 'dec) (set! x (- x 1)))
))))
```

Bank account OOP style

```
(define (new-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
      (begin
        (set! balance (- balance amount))
        balance)
      ”insufficient funds”))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (status) balance)
  (define (dispatch m)
    (cond
      ((eq? m ‘withdraw) withdraw)
      ((eq? m ‘deposit) deposit)
      ((eq? m ‘status) status)
      (else (error “unknown request” m)
    )))
  dispatch)
```

Bank account usage

```
(define acc (new-account 100))
acc => #<procedure:dispatch>

((acc 'withdraw) 20) => 80
((acc 'withdraw) 10) => 70
((acc 'deposit) 40) => 110
((acc 'status)) => 110
```

Sending messages

```
(define (send object msg)
  (if (null? (cdr msg))
    ((object (car msg)))
    ((object (car msg)) (cadr msg))))

(send acc '(withdraw 30)) => 90
(send acc '(deposit 10)) => 100
(send acc '(status)) => 100
```

Different accounts

```
;; two separate accounts
(define jean-acc (new-account 1000))
(define pierre-acc (new-account 1000))

(send jean-acc '(withdraw 100)) => 900
(send tom-acc 'status) => 900

(send pierre-acc 'status) => 1000


;; two shared accounts
(define francois-acc (make-account 1000))
(define laurent-acc francois-acc)

(send francois-acc '(withdraw 100)) => 900
(send francois-acc 'status) => 900
(send laurent-acc 'status) => 900
```

Sequences as Streams

- Modeling the world:
  - model real-world objects with local state by: computational objects with local variables
  - time variation in the real-world by: time variation in the computer
- Reflection: represent time-varying behavior of quantity of x as a function of time x(t)
  - instant by instant: x is a changing quantity
  - mathematically: the function does not change!

Stream

- Streams
  - simply a sequence
    - so we can use lists
  - but shines when using delayed evaluation

Using lists for streams

- Pro: sophisticated procedures available
- Con: severe time and space penalty
- Example: compare:

```
(define (sum-primes a b)
   (define (iter count accum)
      (cond  ((> count b) accum)
             ((prime? count)
               (iter (+ count 1)(+ count accum)))
             (else (iter (+ count 1) accum))))
   (iter a 0))
```

with:

```
(define (list-sum-primes a b)
  (accumulate + 0
         (filter prime? (enumerate-interval a b))))
```

Stream Idea

- use sequence manipulations without incurring the cost of manipulating sequences as lists.
- Idea: construct streams partially, and pass these partial constructs to the program that consumes the stream
  - when the consumer needs part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself
  - maintains illusion that it completely exists

Stream implementation

```scheme
;;Later on we define stream-cons, stream-car, stream-cdr such that
;;  (stream-car (stream-cons x y)) = x
;;  (stream-cdr (stream-cons x y)) = y

(define the-empty-stream '())

(define (stream-null? str)
  (eqv? the-empty-stream str))

(define (stream-ref s n)
   (if   (= n 0)
         (stream-car s)
         (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
   (if   (stream-null? s)
         the-empty-stream
         (cons-stream (proc (stream-car s))
         (stream-map proc (stream-cdr s)))))
```

Stream implementation (ctd)

```
(define (stream-for-each proc s)
   (if   (stream-null? s)
         'done
         (begin   (proc (stream-car s))
                  (stream-for-each proc (stream-cdr s)))))

(define (stream-accumulate op initial str)
  (if (stream-null? str)
      initial
      (op (stream-car str)
          (stream-accumulate op initial (stream-cdr str)))))

(define (stream-filter predicate str)
  (cond ((stream-null? str) '())
        ((predicate (stream-car str))
         (cons-stream (stream-car str)
               (stream-filter predicate (stream-cdr str))))
        (else (stream-filter predicate (stream-cdr str)))))

(define (display-stream s)
   (stream-for-each display-line s))
(define (display-line x)
   (newline) (display x))
```

stream-cons, -car, -cdr

```
(define (stream-car stream)          ;;just return car
   (car stream))

(define (stream-cdr stream)          ;;force evaluation of rest when needed
   (force (cdr stream)))


(define-macro cons-stream            ;;delay evaluation until needed
  (lambda (a b)
    `(cons ,a (delay ,b))))


(define-macro delay                  ;;return promise for exp
   (lambda (expr)
      `(lambda () ,expr)))

(define (force delayed-object)       ;;force delay to fulfill promise
   (delayed-object))
```

Examples

```
(stream-enumerate-interval 15 20) => (15 . #<struct:promise>)

(display-stream (stream-enumerate-interval 15 20)) =>
15
16
17
18
19
20

(display-stream
   (stream-filter (lambda (x) (>= x 18))
   (stream-enumerate-interval 15 20)))
18
19
20

(define (stream-sum-primes a b)
  (stream-accumulate +
              0
              (stream-filter prime? (stream-enumerate-interval a b))))
```

Memoizing

```
;;regularly the same delayed objects are forced many times
;;memoize evaluation in delay

(define (memo-proc proc)
   (let ((already-run? false) (result false))
      (lambda ()
         (if (not already-run?)
            (begin   (set! result (proc))
                     (set! already-run? true)
                     result)
            result))))

(define-macro my-delay
  (lambda (expr)
   `(memo-proc (lambda () ,expr))))
```

Infinite Streams

- Streams can represent sequences that are infinetely long
- Example:

```
(define (integers-from n)
   (cons-stream n (integers-from (+ n 1))))

(define integers (integers-from 1))
```

Infinite Stream Manipulation

- Infinite streams can then be manipulated efficiently
- Example

```
(define (divisible? x y)
   (= (remainder x y) 0))

(define no-sevens
   (stream-filter (lambda (x)
                    (not (divisible? x 7)))
                  integers))

(stream-ref no-sevens 100) => 117
```

Stepping Back...

- Remember:

```
(define (make-withdraw balance)
   (lambda (amount)
       (set! balance (- balance amount))
       balance))
```

- With streams:

```
(define (stream-withdraw balance amount-stream)
   (cons-stream
      balance
      (stream-withdraw
         (- balance (stream-car amount-stream))
         (stream-cdr amount-stream))))
```

- Stream version has no assignment and no state!!
  - looks like having state for user!!

Ramifications

- Functional version does not appear to change over time
  - compare with iterative approach
  - cfr. relativity in physics
- But problems arise when sharing banc-accounts
  - same as for imperative programming

Wrap-up

- Scheme's syntax and evaluation allows for extending the language easily using macroes
- Imperative programming poses serious problems
  - but promotes modularity
- Lazy evaluation
  - is easy to get in Scheme
  - circumvents problems of assignment and state
  - allws to manipulate (infinite) sequences easily

References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/