CSC 533: Programming Languages

Spring 2016

Advanced Scheme programming

- large-scale pattern matching
  Eliza program
- first class functions
  functions as inputs/outputs, lambda expressions
- lazy evaluation
  delay/force, lazy algorithms, lazy lists (streams)
- closures
- OOP in Scheme

1

Large-scale pattern matching

one of the most famous/infamous programs from early AI was Eliza

- simulated an interaction with a psychiatrist
- actually had no language understanding, just a large dataset of question patterns with corresponding responses

```
> (eliza)
Eliza>(hello there)
(how do you do. please state your problem)
Eliza>(i had a bad dream last night)
(what does this dream suggest to you)
Eliza>(that i am afraid of something)
(do you want to be afraid of something)
Eliza>(no)
(are you saying no just to be negative)
Eliza>(i hate to be afraid)
(please continue)
Eliza>(what should i do)
(what comes to your mind when you ask that)
Eliza>(my mother)
(tell me more about your family)
```

see eliza.ss in code directory

2

Merging two lists

consider the task of merging two sorted list (e.g., as part of merge sort)

```
(define (merge list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        ((< (car list1) (car list2))
         (cons (car list1) (merge (cdr list1) list2)))
        (else (cons (car list2) (merge list1 (cdr list2))))))
```

- e.g., have two large lists of sorted golf scores & want to merge

```
➤ (define scores1 '(67 68 69 71 72 72 72 73 73 74))
➤ (define scores2 '(66 68 69 72 74 74 75 75 76 80 81 84))
➤ (merge scores1 scores2)
(66 67 68 68 69 69 71 72 72 72 72 73 73 74 74 74 75 75 76 80 81 84)
```

3

Generalizing the merge

this works fine if you have a list of numbers, but what about different types?

- e.g., lists of strings

```
(define names1 '("Alex" "Betni" "Brian" "Dave" "Mark" "Tim"))

(define names2 '("Colin" "Fletcher" "Jesse" "Juan" "Justin" "Kifton"))
```

- < only works on numbers, we need to use `string<?`

```
(define (merge list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        ((string<? (car list1) (car list2))
         (cons (car list1) (merge (cdr list1) list2)))
        (else (cons (car list2) (merge list1 (cdr list2))))))
```

- we would need a different version for every type of list!

4

Functions as inputs

better solution – parameterize

- functions can be passed as input to other functions
- here, pass the comparison function to `merge`

```
(define (merge func list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        ((func (car list1) (car list2))
         (cons (car list1) (merge (cdr list1) list2)))
        (else (cons (car list2) (merge list1 (cdr list2))))))
```

```
➤(merge < scores1 scores2)
(66 67 68 68 69 69 71 72 72 72 72 73 73 74 74 74 75 75 76 80 81 84)

➤(merge string<? names1 names2)
("Alex" "Betni" "Brian" "Colin" "Dave" "Fletcher" "Jesse" "Juan"
"Justin" "Kifton" "Mark" "Tim")
```

5

Generalizing the merge

but what if data in the list is more complex?

```
(define map1 '((Smith J 67) (Walker T 68) (Fjell M 69) (Kelly K 71)
               (Brown N 72) (Smalls J 72) (Edwards J 72) (Owens A 73)
               (Green H 73) (Cho J 74)))

(define map2 '((Allen M 66) (Rodriguez J 68) (Wills K 69) (Sams C 72)
               (Miller C 74) (Malik S 74) (Ellis B 75) (Evans C 75)
               (Paul J 76) (Reges S 80) (Jefferson T 81) (Woods E 84)))
```

- here, need a function for comparing lists using the # in index 2

```
(define (compIndex2 player1 player2)
  (< (caddr player1) (caddr player2)))
```

```
➤(compIndex2 '(Smith J 72) '(Woods E 84))
#t

➤(merge compIndex2 map1 map2)
((Allen M 66) (Smith J 67) (Rodriguez J 68) (Walker T 68) . . . (Woods E 84))
```

6

Anonymous functions

defining `compIndex2` seems like overkill – will it every be used again?

- Scheme allows for defining anonymous (unnamed) functions using `lambda`
- based on the lambda calculus of Alonzo Church

```
(lambda (in1 in2 … inN) output)
```

defines an anonymous function, mapping in1, in2,…,inN to output

- note: the old Scheme style of defining a function used lambda

```
(define (incr x)            (define incr
  (+ x 1))                    (lambda (x) (+ x 1)))
```

```
➢(merge (lambda (p1 p2) (< (caddr p1) (caddr p2))) map1 map2)
((Allen M 66) (Smith J 67) (Rodriguez J 68) (Walker T 68) . . . (Woods E 84))
```

- suppose the data was structured differently: `((67 Smith J) (68 Walker T) … )`

```
➢(merge (lambda (p1 p2) (< (car p1) (car p2))) map1 map2)
((66 Allen M) (67 Smith J) (68 Rodriguez J) (68 Walker T) . . . (84 Woods E))
```

7

Generalizing the comparison

suppose we needed to do a lot of list comparisons

- the type of value and index for comparison may vary

```
(lambda (p1 p2) (< (list-ref p1 2) (list-ref p2 2)))

(lambda (p1 p2) (string<? (list-ref p1 0) (list-ref p2 0)))
```

- again, we could parameterize

```
(define (compare func index p1 p2)
  (func (list-ref p1 index) (list-ref p2 index)))
```

```
➢(compare < 2 '(Smith J 72) '(Woods E 84))
#t

➢(compare string<? 1 '(J Smith) '(E Woods))
#t
```

8

Functions as outputs

unfortunately, the merge function expects a comparison function with 2 inputs

- we can get around this by having a function that returns a function

```
(define (compareGen func index)
  (define (compare v1 v2)
    (func (list-ref v1 index) (list-ref v2 index)))
  compare)
```

- or, equivalently

```
(define (compareGen func index)
  (lambda (v1 v2)
    (func (list-ref v1 index) (list-ref v2 index))))
```

```
➢(compareGen < 2)
#<procedure:comp>

➢((compareGen < 2) '(Smith J 72) '(Woods E 84))
#t

➢(merge (compareGen < 2) map1 map2)
((Allen M 66) (Smith J 67) (Rodriguez J 68) (Walker T 68) . . . (Woods E 84))
```

9

Merging revisited

suppose you only need to first N items in the lists, for some small N

- e.g., only need the top 10 golf scores to award prizes

```
(define (head arblist len)
  (if (or (null? arblist) (zero? len)) '()
      (cons (car arblist) (head (cdr arblist) (sub1 len)))))
```

```
➢ (define scores1 '(67 68 69 71 72 72 72 73 73 74))
➢ (define scores2 '(66 68 69 72 74 74 75 75 76 80 81 84))
➢ (head (merge scores1 scores2) 10)
(66 67 68 68 69 69 71 72 72 72)
```

WASTEFUL – why merge the remaining scores when not needed?

10

Lazy evaluation

Scheme provides built-in functions for performing *lazy evaluation*

- `delay` takes a functional expression and evaluates to a *promise*
  a *promise* is an object that encapsulates the functional expression (but does not evaluate it)
- `force` takes a promise and executes it, evaluating to the expression value

```
➢ (delay (+ 3 2))
#<promise:unsaved-editor696:10:2>

➢ (define expr (delay (+ 3 2)))

➢ (force expr)
5
```

11

Lazy merging

can modify the merge using `delay` to postpone merges until needed

```
(define (lazyMerge func list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        ((func (car list1) (car list2))
         (cons (car list1)
               (delay (lazyMerge func (cdr list1) list2))))
        (else (cons (car list2)
                    (delay (lazyMerge func list1 (cdr list2)))))))
```

- e.g., lazyMerging two lists returns a pair: `(car. #<promiseOfCdr>)`
- such a structure is known as a *stream*, since it represents a potential stream of data

```
➢ (define stream (lazyMerge scores1 scores2))
(66 . #<promise:...S16/Code/lazyeval.ss:22:32>)
➢ (car stream)
66
➢ (car (force (cdr stream))
67
```

12

Lazy merging (cont.)

need a version of head that uses `force` to expand each cdr promise

```
(define (lazyHead stream len)
  (if (or (null? stream) (zero? len)) '()
      (cons (car stream)
            (lazyHead (force (cdr stream) (sub1 len))))))
```

```
➢ (define scores1 '(67 68 69 71 72 72 72 73 73 74))
➢ (define scores2 '(66 68 69 72 74 74 75 75 76 80 81 84))
➢ (lazyHead (lazyMerge scores1 scores2) 10)
(66 67 68 68 69 69 71 72 72 72)
➢ (lazyHead (lazyMerge scores1 scores2) 15)
(66 67 68 68 69 69 71 72 72 72 72 73 73 74 74)
```

to get the first N values, only do N comparisons/cons operations

Infinite streams?

since streams don't generate the cdr until needed, can be infinite!

```
(define (arithSequence start step)
     (cons start (delay (arithSequence (+ step start) step))))
```

```
➢ (define ones (arithSequence 1 0))
➢ (lazyHead ones 5)
(1 1 1 1 1)

➢ (define naturals (arithSequence 1 1))
➢ (lazyHead naturals 10)
(1 2 3 4 5 6 7 8 9 10)
```

EXERCISE: define infinite streams of even numbers. then odd numbers.

More exercises

define a function for generating geometric sequences

```
(define (geoSequence start factor)
     ( ??? )
```

then, define a stream that represents the infinite geometric sequence made up of powers of 2 (i.e., 1, 2, 4, 8, 16, 32, 64, 128, …)

```
➢ (define powersOf2 ( ??? ))

➢ (lazyHead powersOf2 11)
(1 2 4 8 16 32 64 128 256 512 1024)
```

15

Manipulating infinite streams

a stream is an object that represents and infinite sequence

- can define operations similar to standard lists
- but, it only performs operations as needed

```
(define (list-refStream index stream)
  (if (zero? index)
      (car stream)
      (list-refStream (sub1 index) (force (cdr stream)))))


(define (memberStream item stream)
  (cond ((null? stream) #f)
        ((equal? item (car stream)) stream)
        (else (memberStream item (force (cdr stream))))))
```

16

Combining infinite streams

you can combine infinite streams to produce new (infinite) streams

```
(define (combineStreams op stream1 stream2)
  (cons (op (car stream1) (car stream2))
        (delay (combineStreams op (force (cdr stream1)) (force (cdr stream2))))))
```

```
➢ (lazyHead (combineStreams + ones powersOf2) 10)
(2 3 5 9 17 33 65 129 257 513)
```

```
➢ (lazyHead (combineStreams * odds evens) 6)
(2 12 30 56 90 132)
```

```
(define (mergeStreams func stream1 stream2)
  (if (func (car stream1) (car stream2))
      (cons (car stream1)
            (delay (mergeStreams func (force (cdr stream1)) stream2)))
      (cons (car stream2)
            (delay (mergeStreams func stream1 (force (cdr stream2)))))))
```

```
➢ (lazyHead (mergeStreams < odds evens) 8)
(1 2 3 4 5 6 7 8)
```

17

Filtering infinite streams

can even filter out elements of a stream using a filter function

```
(define (filterStream filter stream)
  (if (filter (car stream))
      (cons (car stream) (delay (filterStream filter (force (cdr stream)))))
      (filterStream filter (force (cdr stream)))))
```

```
➢ (lazyHead (filterStream even? naturals) 10)
(2 4 6 8 10 12 14 16 18 20)
```

```
➢ (lazyHead (filterStream odd? naturals) 10)
(1 3 5 7 9 11 13 15 17 19)
```

- can filter the stream using any function

```
➢ (define no3s (filterStream (lambda (x) (> (remainder x 3) 0)) naturals))
```

```
➢ (lazyHead no3s 20)
(1 2 4 5 7 8 10 11 13 14 16 17 19 20 22 23 25 26 28 29)
```

18

Sieve of Eratosthenes

the idea of filtering leads to an efficient algorithm for generating primes

- start with (2 3 4 5 6 7 8 9 10 11 ...)
- first prime number is 2 – filter out all multiples of 2: (2 3 5 7 9 11 13 15 17 19 ...)
- next prime number is 3 – filter out all multiples of 3: (2 3 5 7 11 13 17 19 23 25 29 ...)
- next prime number is 5 – filter out all multiples of 5: (2 3 5 7 11 13 17 19 23 29 ...)
- . . .

```
(define (sieve stream)
  (cons (car stream)
        (delay (sieve (filterStream (lambda (x) (> (remainder x (car stream)) 0))
                                    (force (cdr stream)))))))
```

```
➢ (define primes (sieve (force (cdr naturals))))
➢ (lazyHead primes 20)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71)
➢ (list-refStream 1000 primes)
7927
```

Static scoping

recall that Scheme is statically scoped

- this is useful for hiding helper functions

```
(define (lengthTail arblist)
  (define (lengthHelp sublist sofar)
    (if (null? sublist)
        sofar
        (lengthHelp (cdr sublist) (add1 sofar))))
  (lengthHelp arblist 0))
```

- can also access other variables that exist within the scope

```
(define (sumToTail N)
  (define (sumHelp curr sumSofar)
    (if (> curr N)
        sumSofar
        (sumHelp (add1 curr) (+ curr sumSofar))))
  (sumHelp 1 0))
```

Functions & scoping

recall the comparison generator function from earlier

- note that inner function uses `func` and `index`, parameters to outer function
- OK since `compare` is defined within the scope of `func/index`

```
(define (compareGen func index)
  (define (compare v1 v2)
    (func (list-ref v1 index) (list-ref v2 index)))
  comp)
```

- but the returned function persists after the call to `compareGen` ends!
- is this a dangling reference problem?

- Scheme handles this by encapsulating the function with its current scope
  essentially, the independent function carries its environment with it

Closures

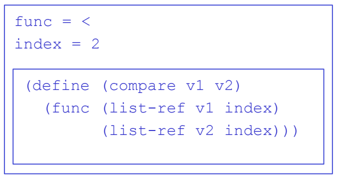
a *closure* is a persistent scope

- here, the inner function `compare` is defined within the scope of `index`
- when the `compareGen` function returns the `compare` function itself, the scope that contains `index` is encapsulated and returned with it

```
(define (compareGen func index)
  (define (compare v1 v2)
    (func (list-ref v1 index) (list-ref v2 index)))
  comp)
```

```
➢ (define comp (compareGen < 2))

➢ (comp '(a b 8) '(c d 4))
#f
```



```
func = <
index = 2

(define (compare v1 v2)
  (func (list-ref v1 index)
        (list-ref v2 index)))
```

Mystery function

what does the following function do?

```
(define (lock message password)
  (define (unlock pass)
    (if (equal? pass password)
        message
        "SORRY, IT'S PRIVATE."))
  unlock)
```

```
(define secret
   (lock "Hi there" "foobar"))
```

```
➢  (secret "barfoo")
"SORRY, IT'S PRIVATE."

➢  (secret "foobar")
"Hi there"
```

```
message = "Hi there"
password = "foobar"

 (define (unlock pass)
   (if (equal? pass password)
       message
       "SORRY"))
```

23

OOP in Scheme

closures allow for data encapsulation & information hiding

- the variables defined in the closure are not accessible outside of that scope
- they are essentially private to the closure function

we can take advantage of this to do OOP in Scheme

example: bank account

data:	account balance
operations:	initialize with some amount
	deposit some amount
	withdraw some amount

24

Naïve (imperative) solution

- use global variable to represent the balance
- initialize and update the balance using `set!`

```
(define balance 100)

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

(define (deposit amount)
  (begin (set! balance (+ balance amount)) balance))
```

```
➢ (withdraw 25)
75
➢ (deposit 50)
125
➢ (withdraw 200)
"Insufficient funds"
```

DRAWBACKS
- no encapsulation
- no data hiding
- not easily extended to multiple accounts

OOP behavior

following OOP principles, would like the following behavior

`(define savings (account 100))`	creates an account called savings, initialized to $100
`(savings 'deposit 50)`	updates the savings account by depositing $50
`(savings 'withdraw 50)`	updates the savings account by withdrawing $50

want balance to be inaccessible except through deposit & withdraw

SOLUTION: make an account object be a *function*
- contains the balance as part of its closure
- recognizes deposit and withdraw commands as input

OOP solution

```
(define (account balance)

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

  (define (deposit amount)
    (begin (set! balance (+ balance amount)) balance))

  (define (menu message arg)
    (cond ((equal? message 'deposit) (deposit arg))
          ((equal? message 'withdraw) (withdraw arg))
          ((else "Unknown operation"))))
  menu)
```

since the returned function is in the scope of the balance parameter and the other inner functions, that parameter and functions are encapsulated into the closure

`(savings 'deposit 50)` applies the menu function to the arguments

27

OOP analysis

this implementation provides

- encapsulation: balance & operations are grouped together
- info hiding: balance is hidden in an account object, accessible via ops

can have multiple objects – each has its own private balance

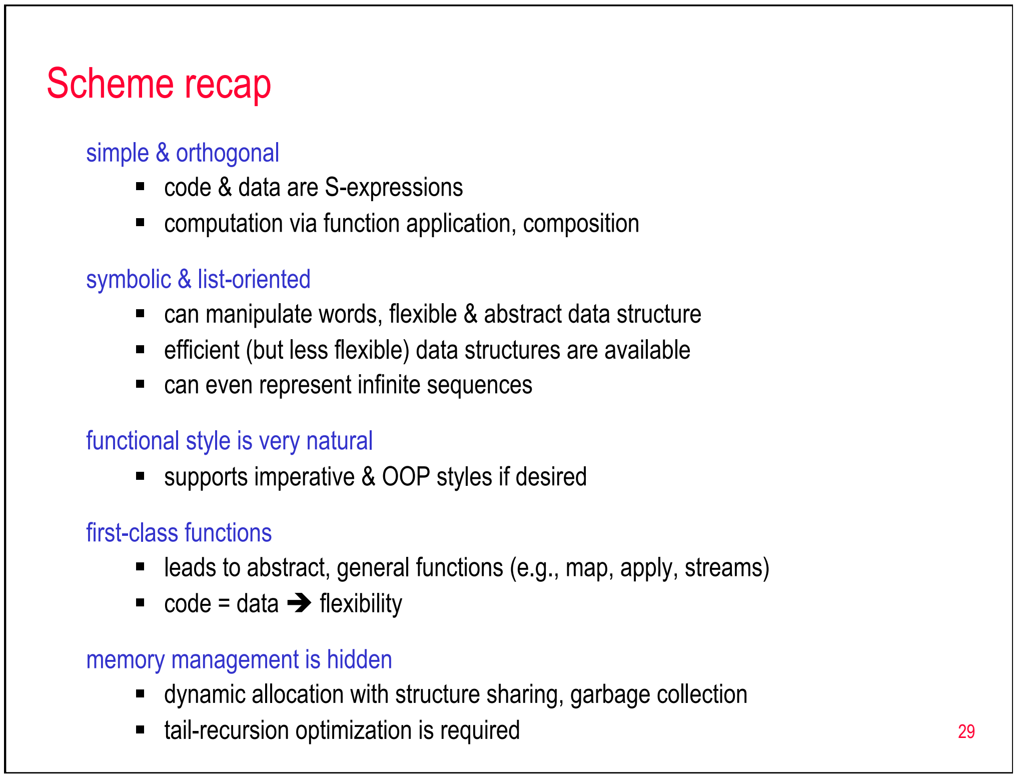
```
(define checking (account 100))
(define savings (account 500))

(checking 'withdraw 50)
(savings 'deposit 50)
```

note: this notation can become a bit awkward

- most Schemes provide an OOP library that insulates the user from details
- allows more natural definitions, inheritance, . . .

28



Scheme recap

simple & orthogonal
- code & data are S-expressions
- computation via function application, composition

symbolic & list-oriented
- can manipulate words, flexible & abstract data structure
- efficient (but less flexible) data structures are available
- can even represent infinite sequences

functional style is very natural
- supports imperative & OOP styles if desired

first-class functions
- leads to abstract, general functions (e.g., map, apply, streams)
- code = data ➔ flexibility

memory management is hidden
- dynamic allocation with structure sharing, garbage collection
- tail-recursion optimization is required

29