Programming Languages

Scheme part 4 and other functional

2020

Instructor: Odelia Schwartz

Lots of equalities!

Summary:

- eq? for symbolic atoms, not numeric (eq? ‘a ‘b)
- = for numeric, not symbolic (= 5 7)
- eqv? for numeric and symbolic

equal versus equalsimp

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((eq? (car lis1) (car lis2))
         (equalsimp (cdr lis1)
(cdr lis2)))
    (else #f)
  )
)
```

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1
lis2))
    ((not (list? lis2)) #f)
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((equal (car lis1) (car
lis2))
         (equal (cdr lis1) (cdr
lis2)))
    (else #f)
  )
)
```

append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- Reminding ourselves of cons (run it on csi):

(cons ‘(a b) ‘(c d)) returns ((a b) c d)

(cons ‘((a b) c) ‘(d (e f))) returns (((a b) c) d (e f))

Adding a list of numbers

- This works: (+ 3 7 10 2)
- This doesn't work: (+ (3 7 10 2))
- Why?

Adding a list of numbers

- This works: (+ 3 7 10 2)
- This doesn’t work: (+ (3 7 10 2))

How would we achieve the second option?

Adding a list of numbers

- This works: (+ 3 7 10 2)
- This doesn’t work: (+ (3 7 10 2))

How would we achieve the second option?

Breakout groups

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

We'll do a little "trick" …

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- cons creates new list with + and a_list

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- cons creates new list with + and a_list

- Why the quote on `+?

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- cons creates new list with + and a_list
- Why the quote on `+?
- Quote so that eval will not evaluate in evaluation of cons

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- Adder (+ 1 2 3 4)
- Calls (eval (+ 1 2 3 4))
- And returns (+ 1 2 3 4)

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- Create adder function and load into csi

- Run on sci adder (+ 1 2 3 4)

- Run on sci (eval (+ 1 2 3 4))

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

Examples:

(adder '(1 2 3))

Adding a list of numbers

- We want: (+ (3 7 10 2))

Let’s each write another way of doing this…

Create adder2 function and load into csi

Run on sci (adder2 ‘(3 7 10 2))

Adding a list of numbers

- We want: (+ (3 7 10 2))

Let’s each write another way of doing this…
Hint: use car and cdr

Create adder2 function and load into csi

Run on sci (adder2 ‘(3 7 10 2))

Other functional languages

Common LISP

- Combination of many features of popular dialects of LISP, early 1980s
- Large and complex language, opposite of Scheme
- Features include: records; arrays; complex numbers; character strings; iterative control statements; etc.
- So not purely functional, has imperative features

Common LISP

- Combination of many features of popular dialects of LISP, early 1980s

Common LISP

- Combination of many features of popular dialects of LISP, early 1980s
- Large and complex language, opposite of Scheme

Common LISP

- Combination of many features of popular dialects of LISP, early 1980s
- Large and complex language, opposite of Scheme
- Features include: records; arrays; complex numbers; character strings; iterative control statements; etc.

Common LISP

- Combination of many features of popular dialects of LISP, early 1980s
- Large and complex language, opposite of Scheme
- Features include: records; arrays; complex numbers; character strings; iterative control statements; etc.
- So not purely functional, has imperative features

ML Language

- Syntax closer to Pascal and other imperative than to LISP

ML Language

- Syntax closer to Pascal and other imperative than to LISP

- Strongly typed (whereas Scheme is essentially typesless) with **no type coercions**

  What were those?

ML Language

- Syntax closer to Pascal and other imperative than to LISP
- Strongly typed (whereas Scheme is essentially typesless) with **no type coercions**
- Has identifiers, but once set cannot be changed – more like final declarations in Java or const in C/C++

Functional declarations ML

- Format:

  fun name (parameters) = body;

Functional declarations ML

- Format:

  fun name (parameters) = body;

Example (run it):

```
fun circumf(r) = 3.14*r*r;
```

Functional declarations ML

- Format:

  fun name (parameters) = body;

Example:

```
fun circumf(r) = 3.14*r*r;
```

The type here is inferred as **float** from the type of the literal in the expression

Functional declarations ML

- Format:

  fun name (parameters) = body;

Example:

fun times10(x) = 10*x;

Inferred as **int**

Functional declarations ML

- Format:

  fun name (parameters) = body;

Example:

fun square(x) = x*x;

Also inferred as **int** (default type)

Functional declarations ML

- Format:

  fun name (parameters) = body;

Example:

fun square(x) = x*x;

Also inferred as **int** (default type)

What happens if called with square(2.75)???

ML Language

https://www.tutorialspoint.com/execute_smlnj_online.php

ML Language

- Try running some code:

```
fun times10(x) = 10*x;
times10(5);
```

ML Language

- Try running some code:

```
fun times10(x) = 10*x;
times10(5);

times10(5.1);
```

What happens???

ML Language

- Try running some code:

```
fun times10(x) = 10*x;
times10(5);

times10(5.1);
```

Yields error; expecting int…

It’s strongly typed!!

ML Language

- We could also specify type.

```
fun square(x:real) = x * x;
```

ML Language

- We could also specify type.

```
fun square(x:real) = x * x;
```

Enough to infer that type is real

ML Language

- These are all valid:

```
fun square(x:real) = x * x;

fun square(x) = (x:real) * x;

fun square(x) = x * (x:real)
```

Enough to infer that type is real

ML Language

- These are all valid:

```
fun square(x:real) = x * x;

fun square(x) = (x:real) * x;

fun square(x) = x * (x:real)
```

Enough to infer that type is real

Type inference also used in Haskell, Miranda, F#

ML Language

- Try running some more code:

```
fun square(y:real) = y*y;
square(5.1);
square(5.0);
```

ML Language

- What about this?

```
fun square(y:real) = y*y;
square(5);
```

ML Language

- What about this?

```
fun square(y:real) = y*y;
square(5);
```

Oops another type error…

ML Language

- What about this?

```
fun square(y:real) = y*y;
square(5);
```

Oops another type error…

Note: user defined overloaded functions not allowed, so if we wanted a square function, one for real and one for int, would have to use different names…

ML selection

- if else format:

  if expression then expression

  else else_expression

ML selection

- Example:

```
fun fact (n:int) =

if n<=1 then 1
else n * fact(n-1);
```

ML selection

- Example:

```
fun fact (n:int) =

if n<=1 then 1
else n * fact(n-1);

fact(4);
```

Run it…

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
```

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
| fact(1) = 1
```

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
| fact(1) = 1
| fact(n:int) = n*fact(n-1);
```

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
| fact(1) = 1
| fact(n:int) = n*fact(n-1);
```

Meant to mimic conditional functional definitions in math…

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
| fact(1) = 1
| fact(n:int) = n*fact(n-1);
```

Meant to mimic conditional functional definitions in math…

If param is int that is not 0 or 1 then third definition is used…

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
| fact(1) = 1
| fact(n:int) = n*fact(n-1);
```

Note that don't need the **int** here since it is the default

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
| fact(1) = 1
| fact(n) = n*fact(n-1);
```

So this is also OK

ML selection

- Another way: pattern matching!

```
fun fact(0) = 1
| fact(1) = 1
| fact(n) = n*fact(n-1);
fact(4)
```

Let’s try running code above…

ML list operations

- hd, tl are ML's version of Scheme CAR, CDR

ML list operations

- hd, tl are ML's version of Scheme CAR, CDR
- Literal lists in brackets [3,5,7]; [] empty list

ML list operations

- hd, tl are ML’s version of Scheme CAR, CDR
- Literal lists in brackets [3,5,7]; [] empty list
- :: used for cons

ML list operations

- hd, tl are ML's version of Scheme CAR, CDR
- Literal lists in brackets [3,5,7]; [] empty list
- :: used for cons

4::[3,5,7] evaluates to?

ML list operations

- hd, tl are ML's version of Scheme CAR, CDR
- Literal lists in brackets [3,5,7]; [] empty list
- :: used for cons

4::[3,5,7] evaluates to?

[4,3,5,7]

ML list operations

- Try running these:

```
4::[3,5,7]

hd([4,3,5,7])

tl([4,3,5,7])
```

ML list operations

- Number of elements in a list

```
fun length([]) = 0
```

ML list operations

- Number of elements in a list

```
fun length([]) = 0
| length(h::t) = 1 + length(t);
```

ML list operations

- Number of elements in a list

```
fun length([]) = 0
| length(h::t) = 1 + length(t);
length([1,3,5])
```

Try running it

ML list operations

- Append function

fun append ([],lis2) = **?**

(what should we write here?)

ML list operations

- Append function

```
fun append ([],lis2) = lis2
```

ML list operations

- Append function

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = ?
```

What should we do?

ML list operations

- Append function

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = h::?
```

ML list operations

- Append function

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = h::append(t,lis2);
```

ML list operations

- Append function

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = h::append(t,lis2);
append([1,2],[3,4]);
```

Try running it…

Let’s remind ourselves Scheme

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- Reminding ourselves of cons (run it on csi):

(cons ‘(a b) ‘(c d))   returns ((a b) c d)

(cons ‘((a b) c) ‘(d (e f)))   returns (((a b) c) d (e f))

ML versus Scheme append

```
fun append ([],lis2) = lis2
| append(h::t,lis2) =
h::append(t,lis2);
```

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

ML list operations

- Let's each try fun adder

  adder([1,2,3]) should return 6

ML list operations

- Let's each try fun adder

```
fun adder([]) = 0
| adder (h::t)=h+adder(t);
adder([1,2,3,4,5]);
```

Names bound to values (constants)

- Format:

val new_name = expression;

Names bound to values (constants)

- Format:

```
val new_name = expression;
```

Binds the value to name once and cannot be rebound (nothing like an assignment statement in an imperative language!)

Names bound to values (constants)

- Format:

```
val new_name = expression;
```

Example: usually used with a let statement:

```
fun area(radius) =
let val radius = 2.7
    val pi = 3.14159
in pi*radius*radius
end;
```

Higher order functions

- map

```
map(fn x =>x*x*x)[1,3,5];
```

Higher order functions

- map

```
map(fn x =>x*x*x)[1,3,5];
```

Note: different interpreters have slightly different notation; book notation different

Higher order functions

- Composing two functions

h = f o g

(lower case o)

Higher order functions

- Composing two functions

h = f o g

Example: (run it)

```
fun times10(x) = 10*x;
times10(5);
fun plus3(y) = 3 + y;
plus3(4);
val h = times10 o plus3;
h(7)
```