EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

November 3rd, 2015

KU

Is this a valid scheme command?

```
> (1 + (2 * 3))
```

KU

Is this a valid scheme command?

```
> (1 + (2 * 3))
```

No.

```
expected procedure, given: 2; arguments were: #<procedure:*> 3
```

Is this a valid scheme command?

```
> '(1 + (2 * 3))
```

Is this a valid scheme command?

```
> '(1 + (2 * 3))
```

Yes! - it is a quoted S-Expression

```
(1 + (2 * 3))
```

This looks like an arithmetical expression, but is actually Scheme data.

We want to write a scheme function, `value`

```
> (value '(1 + (2 * 3)))
7
> (value '(2 * 3))
6
```

value takes a (quoted) expression, and returns a number.

Assumptions: the input is well formed

```
E ::=  number
     | ( E + E )
     | ( E * E )
```

What questions do we need to ask?

What questions do we need to ask?

- Are you at atom?
  (must be a number!)
- Are you an addition?
- Are you a multiplication?

KU

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))

(define value
 (lambda (nexp)
  (cond
   ((atom? nexp) nexp)
   ((...) ...)
   ((...) ...)
  ) ... )
```

```
> (value '4)
4
```

KU

```
> (value '(2 + 3))
...
```

We can test for (E + E) using '+

```
(eq? (car (cdr nexp)) '+)
```

```
(define value
 (lambda (nexp)
  (cond
   ((atom? nexp) nexp)
   ((eq? (car (cdr nexp)) '+)
         (+ (value (car nexp))
            (value (car (cdr (cdr nexp))))))
   ((eq? (car (cdr nexp)) '*)
         (* (value (car nexp))
            (value (car (cdr (cdr nexp))))))
  )
 )
)
```

```
> (value '(4 + 3))
7
```

We have taken a quoted S-Expression, and given it meaning!

```
> (value '((1 + 2) * (4 + 3)))
21
> (value '4)
4
> (value '(4))
 car: contract violation
  expected: pair?
  given: '()
```

How do we fix the last example?

Homework!

Write a function which multiplies the height from the root to each element in an s-expresssion.

```
> (timesdepth* 1 '(1 2 3))
(1 2 3)
> (timesdepth* 1 '((1) (2) (3)))
((2) (4) (6))
> (timesdepth* 1 '((1) 2 ((3))))
((2) 2 ((9)))
```

Test out your function with Racket. If you are stuck, write `timesdepth`, that works on a list of atoms. You can assume that the atoms are numbers.

KU

Lambda the Ultimate

KU

eq? and equal? and =

			eq?	equal?	=
`'a`	`'a`	⇒	`#t`	`#t`	error
`'(a)`	`'(a)`	⇒	`#f`	`#t`	error
`'(a b)`	`(a b)`	⇒	`#f`	`#t`	error
`'(b a)`	`'(a b)`	⇒	`#f`	`#f`	error
`'(a (b) c)`	`'(a (b) c)`	⇒	`#f`	`#t`	error
`1`	`2`	⇒	`#f`	`#f`	`#f`
`1.0`	`1.0`	⇒	`#t`	`#t`	`#t`
`1`	`1`	⇒	`#t`	`#t`	`#t`
`100000`	`100000`	⇒	`#t`	`#t`	`#t`
`10000000000`	`10000000000`	⇒	`#t`	`#t`	`#t`

KU

Keyword-like things and built-in things in scheme

```
define lambda cond else

#t #f

if let begin set!
```

Function-like things in scheme

```
eq? equal? null? zero? cons?

+ - * /

=    <   >   <=   >=

and or not

cons car cdr list

display
```

find index function

```
(define findindex
  (lambda (a l)
          (findindex2 a l 0)))
```

find index sub-function

```
(define findindex2
 (lambda (a l ix)
  (cond
   ((null? l) -1)
   (else
    (cond
     ((= a (car l)) ix)
     (else (findindex2 a (cdr l) (+ 1 ix))))))))
```

passing functions as arguments

```
(define findindex2
 (lambda (cmp a l ix)
  (cond
   ((null? l) -1)
   (else
    (cond
     ((cmp a (car l)) ix)
     (else (findindex2 cmp a (cdr l) (+ 1 ix)))
    )))))

(define findindex-using
  (lambda (cmp a l)
       (findindex2 cmp a l 0)))
```

KU

map

```
(define double
  (lambda (a) (+ a a)))
```

```
> (map double '(1 2 3 4))
```

KU

map

```
(define double
  (lambda (a) (+ a a)))
```

```
> (map double '(1 2 3 4))

(2 4 6 8)
```

map

```
(define double
  (lambda (a) (+ a a)))
```

```
> (map double '(1 2 3 4))

(2 4 6 8)

> (map
    (lambda (a) (* a a))
    '(1 2 3 4))
```

map

```
(define double
  (lambda (a) (+ a a)))
```

```
> (map double '(1 2 3 4))

(2 4 6 8)

> (map
    (lambda (a) (* a a))
    '(1 2 3 4))

(1 4 9 16)
```

KU

foldr

```
(define foldr
   (lambda (f z l)
      (cond
        ((null? l) z)
        (else (f (car l) (foldr f z (cdr l)))))))
```

```
> (foldr + 0 '(1 2 3 4 5))
```

foldr

```
(define foldr
   (lambda (f z l)
      (cond
        ((null? l) z)
        (else (f (car l) (foldr f z (cdr l)))))))
```

```
> (foldr + 0 '(1 2 3 4 5))

15
```

foldr

```
(define foldr
   (lambda (f z l)
      (cond
        ((null? l) z)
        (else (f (car l) (foldr f z (cdr l)))))))
```

```
> (foldr + 0 '(1 2 3 4 5))

15

> (foldr cons '() '(1 2 3 4 5))
```

foldr

```
(define foldr
   (lambda (f z l)
      (cond
        ((null? l) z)
        (else (f (car l) (foldr f z (cdr l)))))))
```

```
> (foldr + 0 '(1 2 3 4 5))

15

> (foldr cons '() '(1 2 3 4 5))

(1 2 3 4 5)
```

KU

Real World Scheme

Scheme in the Real World

So far, we can evaluate small functions, but do no real work.
In this class we will step back, and see some other "features" of scheme.

- Special Forms – (cond ...) (define ...) etc.
- Assignment – (set! ...)
- Scoping Rules
- Input and Output

KU

Special Forms

We have met several special forms already

```
(define xyz ...)
```

```
(lambda (a b c) ...)
```

```
(define add1
   (lambda (x) (+ x 1)))
```

KU

Lambda

```
(define double (lambda (a) (* a 2)))
(double 4)
```

Lambda

```
(define double (lambda (a) (* a 2)))
(double 4)

(    (lambda (a) (* a 2))  4 )
```

Lambda

```
(define double (lambda (a) (* a 2)))
(double 4)

(    (lambda (a) (* a 2))  4 )

(                 * 4 2      )
```

let

```
(let ((v 4)
      (w 9))
      (* v w))
```

KU

let

```
(let ( (double (lambda (a) (* a 2)))
       (v 4)
     )
     (double v)
)
```

let and lambda

```
(     (lambda (x) (* x x))  9 )
```

```
(let ( (x 9)
       )
       (* x x)
)
```

Assignment

```
(define x 1)
(+ x 1)                 ;; ==> 2
(set! x 4)
(+ x 1)                 ;; ==> 5
```

`define` creates a binding for `x`, while `set` changes it.

set! vs. define

```
(define v ..)
```

has essentially the same effect as this assignment expression, if variable is bound:

```
(set! v ...)
```

but is about *namespace*, not assignment!

KU

Grouping

```
(define x 1)
(define f
  (lambda (y)
    (begin
      (set! x y)
      (+ y y))))
(f 99)
```

KU

Static Scoping

```
(define x 1)
(define g (lambda (y) (+ x y)))
(define f (lambda (x) (g 2)))
(f 5)
```

KU

cond

```
(define x 1)
(cond ((= x 1) 99)
      ((= x 9) 100)
      (else 121)
)
```

KU

if

```
(define x 1)
(if (= x 22) 99 100)
```

KU

```
> (display "Hello, World")
Hello, World
> (write "Hello, World")
"Hello, World"
>
```

```
> (set! x (read))
Hello
>
```

This is actually reading a scheme s-expression!

grouping, again

```
> (let ((x ""))
       (begin (set! x (read))
              (write x))
  )
Hello World
```

grouping, again

```
> (let ((x ""))
       (begin (set! x (read))
              (write x))
  )
Hello World
 Hello
```