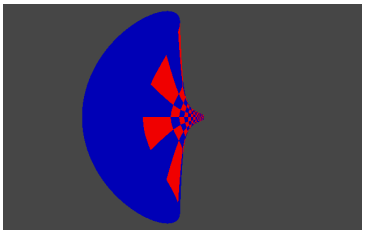
Relativity & Computer Graphics

checkerboard cylinder moving horizontally at 0.9c



Abstraction in the real world

Microsoft windows, Intel processors
- People know too much about low-level detail
- They exploit them and their code does not work on the following version/processor

Graphics card manufacturers
- Extremeley secretive about implementation detail
- Partially for "competition" reasons
- But also, they want a strong abstraction barrier

Dirty if in DrScheme

In practice, "if" accepts non-Boolean arguments

But it is a bad dirty programming habit

Just say "no" to non-Booleans in if

Higher-order procedures in other languages

- Most other languages can take a function as input to some extent
- They rarely can create new functions
  - This is the power of lambda!
- HOP are why we said that procedures are just like any other value
  - (they just have a little more power, but certainly no restriction)

Why type is important

Some languages are typed

It helps you understand HOP

It emphasizes the generality of the pattern

Good sanity check when programming
- A little like units in physics

Typing in languages

Scheme is weakly typed

In contrast to e.g. C++, Java
- But the user has to specify types

In contrast to e.g. ML, Caml
- Also higher-order proc
- Automatic typing
- complex because you need to type procedures

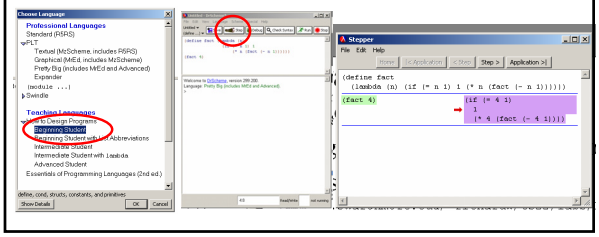
tracing in DrScheme

Only in the Beginner language level

language menu, how to design programs

Press the step button

http://www.cs.swarthmore.edu/~richardw/cs22/labs/1.html



Nano quiz

```
(define incrementby
  (lambda (n)
    (lambda(x) (+ x n))))
(define f1 (incrementby 6))
```

We get a procedure.

```
>(f1 7)
13
(define f1 (lambda(x)(incrementby 6)))
```

This is different, the argument to f1 is just ignored

```
((f1 4) 7)
13
```

(map procedure L)

```
>(map square L)
(1 4 9 16 25 36 49)
```

Type of map?

(a→ b, list of a) → list of b

Implement it yourself

```
(define (map proc lst)
  (if (null? lst) nil
     (cons (proc (car lst))
         (map proc (cdr lst)))))
```

(filter predicate list)

Returns a list with only the elements for which the predicate is true

```
>(define L (list 1 2 3 4 5 6 7))
>(filter even? L)
(2 4 6)
```

Guess the type of filter?

(A → Boolean, list of A) → list of A

Implement filter yourself

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
           (cons (car lst)
                 (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

(foldr proc init L)

(a.k.a fold-right or accumulate)

```
>(foldr + 0 L)
28
```

The <and> of a list of Booleans?

```
(define my-and (lambda(x y) (and x y)))
(foldr my-and true L)
```

As someone pointed out, and is a special form and cannot be used directly here

How do you extract the max of a list?

```
(foldr max (car L) (cdr L))
```

If all numbers of a list are even?

```
(foldr (lambda(x y) (and (even? x) y))
       true L)
```

(foldr proc init L)

Type of foldr?

(((A,B) → B), B, list of A) → B

Implement it.

```
(define (fold-right op init lst)
  (if (null? lst) init
     (op (car lst)
         (fold-right op init (cdr lst)))))
```

recap: List HOP

`(filter predicate L)` to prune list
`(map op L)` when output is a list
`(foldr op init L)` when output is one value "summarizing" the list
Use as much as possible, except when forbidden!
There is also a foldl
guess what it does?
Implement it!

Average of a list

```
(define (average L)
  (/ (foldr + 0 L)
     (length L)))
```

Re-define (length L)

Using one of the list HOP

```
(define (length L)
  (foldr (lambda(x y) (+ 1 y)) 0 L)))
```

Given (define L (list 1 2 3 4 5 6 7)

compute the sum of the squared values

```
(foldr + 0 (map square L))
```

Sum of the even values

```
(foldr + 0 (filter even? L))
```

Range of a list

Difference between biggest and smallest value

```
(- (foldr max (car L) (cdr L))
   (foldr min (car L) (cdr L))
```

Write

`(apply-all list-proc arg)`

that applies a list of functions to the same argument and returns the list of results

```
>(apply-all (list square sqrt) 4)
(16 2)
(define (apply-all list-proc arg)
  (map (lambda(x)(x arg)) L))
```

Write map using foldr

```
(define (map op L)
  (foldr
   (lambda(x subL)
       (cons (op x) subL)
   ()
   L))
```

Write (map2 op L1 L2)

op takes two arguments

```
(define (map2 L1 L2)
  (if (or (null? L1) (null? L2)) ()
    (cons (op (car L1) (car L2))
          (map2 (cdr L1) (cdr L2)))
```

Write (element? x L)

Returns true if x is in L

Use HOP

```
(define (element? x L)
  (not (null?
   (filter (lambda(a)(equal? x a)) L))))
```

Sieve of Eratosthenes

Compute prime numbers using filter

First enumerate the n first integeres

```
(define (enumerate n)
  (define (helper n L)
    (if (= 0 n) L (helper (- n 1) (cons n L))))
  (helper n ()))
```

Define a divisible? helper predicate

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

Next, extract the list of prime numbers by removing multiples

```
(define (sieve L)
  (if (null? L) ()
    (cons (car L)
       (sieve (filter
         (lambda(x)(not (divisible? x (car L))))
         (cdr L))))))
```

Don't forget to remove 1

```
(sieve (cdr (enumerate 100)))
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
  67 71 73 79 83 89 97)
```

Suppose lst is bound to the list (1 2 3 4 5 6 7). Using map, filter, and/or fold-right, write an expression involving lst that returns:

```
(1 4 9 16 25 36 49)
(map square lst)
(1 3 5 7)
(filter odd? lst)
((1 1) (2 2) (3 3) (4 4) (5 5) (6
  6) (7 7))
(map (lambda (x) (list x x)) lst)
((2) ((4) ((6) \#f)))
(fold-right list nil
  (map list (filter even? lst)))
```

Consider the procedure copy which takes a list and returns a copy of the list. How do each of the following differ?

```
(define (copy-ident x) x)
```

This one does not create a new copy. It just points to the old one !

```
(define (copy-recurse x)
  (if (null? x) nil
    (cons (car x)
      (copy-recurse (cdr x)))))
```

This one does the job. It does actually create a new list which contains the same values.

Notice that copy-recurse is a recursive process. Write an iterative copy! Warning, it's tough. See next slide.

Warning: the below is not copy!

```
(define (*copy-iter* x)
  (define (aux x ans)
    (if (null? x) ans
      (aux (cdr x)
           (cons (car x) ans)))
  (aux x nil) )
```

The above is not copy. Actually, it's reverse!
Now let's define copy using reverse:

```
(reverse (reverse x))
```

(append L1 L2)

returns the list composed of all the elements of L1 followed by all the elements of L2

Write yourself "manually"

Discuss tradeoff of what to do with L2. Do you copy it or do you point to it?

Use foldr, map or filter