

Not much new here:

(define w c)

 Environment for closure determined when function is defined, but body is evaluated when function is called

; 7

Once an expression produces a value, it is irrelevant how the value was produced

Autumn 2017

CSE341: Programming Languages

3

5

```
(define pr (cons 1 (cons #t "hi"))); '(1 #t . "hi")
(define lst (cons 1 (cons #t (cons "hi" null))))
(define hi (cdr (cdr pr)))
(define hi-again (car (cdr (cdr lst))))
(define hi-another (caddr lst))
(define no (list? pr))
(define yes (pair? pr))
(define of-course (and (list? lst) (pair? lst)))
```

Passing an improper list to functions like length is a run-time error

Autumn 2017 CSE341: Programming Languages

# The truth about cons

So why allow improper lists?

- Pairs are useful
- Without static types, why distinguish (e1,e2) and e1::e2

Style:

- Use proper lists for collections of unknown size
- But feel free to use cons to build a pair
  - Though structs (like records) may be better

#### Built-in primitives:

- list? returns true for proper lists, including the empty list
- pair? returns true for things made by cons
  - All improper and proper lists except the empty list

Autumn 2017

CSE341: Programming Languages

## cons cells are immutable

What if you wanted to mutate the contents of a cons cell?

- In Racket you cannot (major change from Scheme)
- This is good
  - · List-aliasing irrelevant
  - Implementation can make list? fast since listness is determined when cons cell is created

Autumn 2017

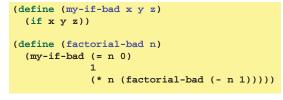
4

### Set! does not change list contents mcons cells are mutable Since mutable pairs are sometimes useful (will use them soon), This does not mutate the contents of a cons cell-Racket provides them too: (define x (cons 14 null)) - mcons (define y x) - mcar (set! x (cons 42 null)) - mcdr (define fourteen (car y)) - mpair? - set-mcar! - Like Java's x = new Cons(42, null), not x.car = 42 - set-mcdr! Run-time error to use mean on a consider of car on an meons cell CSE341: Programming Languages Autumn 2017 CSE341: Programming Languages 7 Autumn 2017 8 Delayed evaluation

For each language construct, the semantics specifies when subexpressions get evaluated. In ML, Racket, Java, C:

- Function arguments are *eager* (call-by-value)
- · Evaluated once before calling the function
- Conditional branches are not eager

It matters: calling factorial-bad never terminates:



Autumn 2017

CSE341: Programming Languages

## Thunks delay

We know how to delay evaluation: put expression in a function! - Thanks to closures, can use all the same variables later

A zero-argument function used to delay evaluation is called a *thunk* - As a verb: thunk the expression

This works (but it is silly to wrap if like this):

```
(define (my-if x y z)
      (if x (y) (z)))
    (define (fact n)
         (my-if (= n 0))
                 (lambda() 1)
                 (lambda() (* n (fact (- n 1)))))
Autumn 2017
                    CSE341: Programming Languages
                                                         10
```

The key point

· Evaluate an expression e to get a result:



 A function that when called, evaluates e and returns result - Zero-argument function for "thunking"

#### (lambda () e)

• Evaluate e to some thunk and then call the thunk

#### (e)

- · Next: Powerful idioms related to delaying evaluation and/or avoided repeated or unnecessary computations
  - Some idioms also use mutation in encapsulated ways

Autumn 2017

CSE341: Programming Languages

11

## Avoiding expensive computations

Thunks let you skip expensive computations if they are not needed

Great if take the true-branch:

(define (f th) (if (...) 0 (... (th) ...)))

But worse if you end up using the thunk more than once:

(define (f th) (... (if (...) 0 (... (th) ...)) (if (...) 0 (... (th) ...)) (if (...) 0 (... (th) ...))))

In general, might not know many times a result is needed Autumn 2017 CSE341: Programming Languages

# Best of both worlds

Assuming some expensive computation has no side effects, ideally we would:

- Not compute it until needed

- *Remember the answer* so future uses complete immediately Called *lazy evaluation* 

Languages where most constructs, including function arguments, work this way are *lazy languages* 

Haskell

Racket predefines support for *promises*, but we can make our own
– Thunks and mutable pairs are enough... [Friday]

Autumn 2017

CSE341: Programming Languages

13