

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

# Programming Languages

Dan Grossman

Datatype-Programming in Racket With Structs

# New feature

```
(struct foo (bar baz quux) #:transparent)
```

Defines a new kind of thing and introduces several new functions:

- `(foo e1 e2 e3)` returns “a foo” with `bar`, `baz`, `quux` fields holding results of evaluating `e1`, `e2`, and `e3`
- `(foo? e)` evaluates `e` and returns `#t` if and only if the result is something that was made with the `foo` function
- `(foo-bar e)` evaluates `e`. If result was made with the `foo` function, return the contents of the `bar` field, else an error
- `(foo-baz e)` evaluates `e`. If result was made with the `foo` function, return the contents of the `baz` field, else an error
- `(foo-quux e)` evaluates `e`. If result was made with the `foo` function, return the contents of the `quux` field, else an error

## *An idiom*

```
(struct const (int) #:transparent)
(struct negate (e) #:transparent)
(struct add (e1 e2) #:transparent)
(struct multiply (e1 e2) #:transparent)
```

For “datatypes” like `exp`, create one struct for each “kind of exp”

- structs are like ML constructors!
- But provide constructor, tester, and extractor functions
  - Instead of patterns
  - E.g., `const`, `const?`, `const-int`
- Dynamic typing means “these are the kinds of exp” is “in comments” rather than a *type system*
- Dynamic typing means “types” of fields are also “in comments”

# *All we need*

These structs are all we need to:

- Build trees representing expressions, e.g.,

```
(multiply (negate (add (const 2) (const 2)))  
          (const 7))
```

- Build our `eval-exp` function (see code):

```
(define (eval-exp e)  
  (cond [(const? e) e]  
        [(negate? e)  
         (const (- (const-int  
                    (eval-exp (negate-e e)))))]  
        [(add? e) ...]  
        [(multiply? e) ...]...)
```

# Attributes

- **`#:transparent`** is an optional attribute on struct definitions
  - For us, prints struct values in the REPL rather than hiding them, which is convenient for debugging homework

- **`#:mutable`** is another optional attribute on struct definitions
  - Provides more functions, for example:

```
(struct card (suit rank) #:transparent #:mutable)  
; also defines set-card-suit!, set-card-rank!
```

- Can decide if each struct supports mutation, with usual advantages and disadvantages
  - As expected, we will avoid this attribute
- `mcons` is just a predefined mutable struct