

```
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 Without Structs

Life without datatypes

Racket has nothing like a datatype binding for one-of types

No need in a dynamically typed language:

- Can just mix values of different types and use primitives like **number?**, **string?**, **pair?**, etc. to “see what you have”
- Can use cons cells to build up any kind of data

This segment: Coding up datatypes with what we already know

Next segment: Better approach for the same thing with structs

- Contrast helps explain advantages of structs

Mixed collections

In ML, cannot have a list of “ints or strings,” so use a datatype:

```
datatype int_or_string = I of int | S of string

fun funny_sum xs = (* int_or_string list -> int *)
  case xs of
    [] => 0
  | (I i) :: xs' => i + funny_sum xs'
  | (S s) :: xs' => String.size s + funny_sum xs'
```

In Racket, dynamic typing makes this natural without explicit tags

- Instead, every value has a tag with primitives to check it
- So just check car of list with **number?** or **string?**

Recursive structures

More interesting datatype-programming we know:

```
datatype exp = Const of int
             | Negate of exp
             | Add of exp * exp
             | Multiply of exp * exp
```

```
fun eval_exp e =
  case e of
    Constant i => i
  | Negate e2 => ~ (eval_exp e2)
  | Add(e1,e2) => (eval_exp e1) + (eval_exp e2)
  | Multiply(e1,e2) => (eval_exp e1) * (eval_exp e2)
```

Change how we do this

- Previous version of `eval_exp` has type `exp -> int`
- From now on will write such functions with type `exp -> exp`
- Why? Because will be interpreting languages with multiple kinds of results (ints, pairs, functions, ...)
 - Even though much more complicated for example so far
- How? [See the ML code file:](#)
 - Base case returns entire expression, e.g., `(Const 17)`
 - Recursive cases:
 - Check variant (e.g., make sure a `Const`)
 - Extract data (e.g., the number under the `Const`)
 - Also return an `exp` (e.g., create a new `Const`)

New way in Racket

See the Racket code file for coding up the same new kind of “**exp** \rightarrow **exp**” *interpreter*

- Using lists where car of list encodes “what kind of exp”

Key points:

- Define our own constructor, test-variant, extract-data functions
 - Just better style than hard-to-read uses of **car**, **cdr**
- Same recursive structure without pattern-matching
- With no type system, no notion of “what is an exp” except in documentation
 - But if we use the helper functions correctly, then okay
 - Could add more explicit error-checking if desired

Optional: Symbols

Will not focus on Racket *symbols* like `'foo`, but in brief:

- Syntactically start with quote character
- Like strings, can be almost any character sequence
- Unlike strings, compare two symbols with `eq?` which is fast