Higher-Order Data
Functions are data

We can:

• pass functions as input
  • map, filter, foldr
• return functions as output
  • (.), partial application
• write function literals
  • \f -> f . f
• store functions in data types!

data Expr
    = Lit Int
    | Ref Var
    | Bin (Int -> Int -> Int) Expr Expr

(BinOp.4.hs)
Dictionary pattern

Basic idea:

• **data type** describes an **interface**
• **values** represent **instances**

*Can do “pure object-oriented programming” in Haskell!*
Combinator pattern

Basic idea:

• **type** defines the **semantic domain** of a language
• **values** represent **semantic objects**
• **functions** define the **syntax** “shallow embedding” of a DSL

A *very* successful pattern in Haskell!

(Env.hs, Hasklet2.hs (Parsers))
Combinator pattern tradeoffs

Advantages:

• easy to extend with new cases and operations
• DSL that reuses features of host language
e.g. bindings, type system
• high compositionality – leads to general, powerful designs

Disadvantages:

• difficult to extend with new interpretations
e.g. pretty print a parser specification
• hard to analyze/manipulate internal structure of values
e.g. optimize a parser
Higher-order abstract syntax pattern

A technique for implementing **name bindings** in ASTs

Basic idea:

- use **host language** (Haskell) bindings to encode **object language** bindings

A “shallow” feature in an otherwise “deep embedding”

Advantage: reuse features of host language

Disadvantages: hard to analyze/manipulate structure

* e.g. optimize, get all variable names