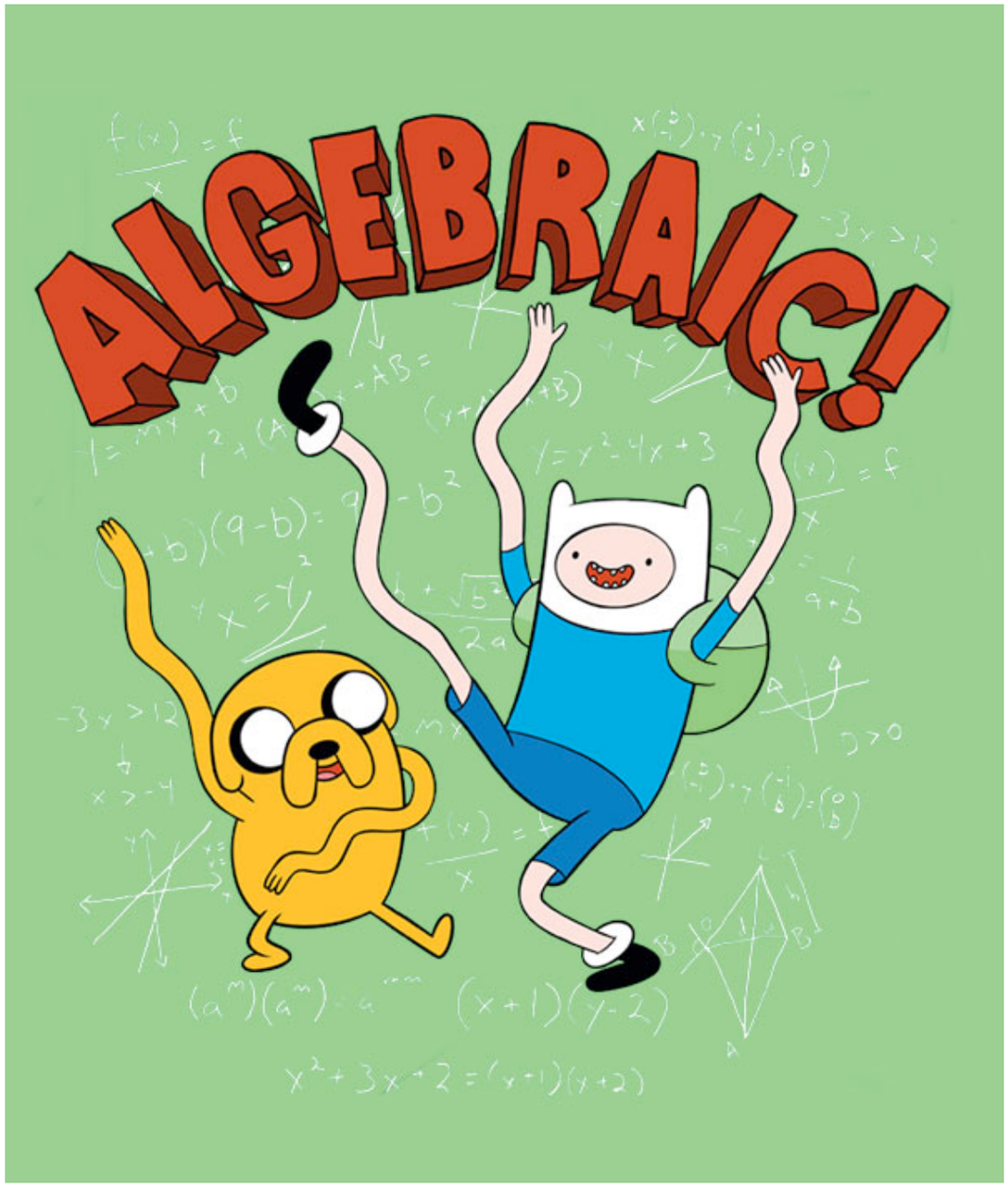
Generalized Algebraic Data Types



Algebraic data types (review)

```
data Expr
   = Lit Int
   | Neg Expr
   | Add Expr Expr
   | Mul Expr Expr
```

Declaring this new data type gives you:

1. *new type* **`Expr`**
2. *several constructors for creating values of type* **`Expr`**

```
Lit :: Int -> Expr
Neg :: Expr -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
```

Limitation of mono-typed expressions

```
data Expr
   -- literals
   = LitI Int
   | LitB Bool
   -- integers
   | Neg Expr
   | Add Expr Expr
   | Mul Expr Expr
-- booleans
| Not Expr
| Or  Expr Expr
| And Expr Expr
-- mixed
| Equ Expr Expr
| If  Expr Expr Expr
```

Problem: can build ill-typed expressions:

```
Add (LitB True) (LitI 4)
```

Solutions:

- *dynamic typing during evaluation*
- *separate type-checking phase*

*Can we use **Haskell's** type system*
*to prevent type errors in the **object language**?*

Getting more out of data types

Tool #1: *phantom types*

- type parameter that isn't an argument to a data constructor
- use to *embed* and *enforce* properties in Haskell types

Tool #2: *generalized algebraic data types*

- write type of each data constructor *explicitly*
  - return types of each data constructor can be different
  - can include class constraints

Very useful for deeply embedded DSLs!

- *embed DSL's type system into Haskell's type system*

Parametrically typed expressions

Idea: add more type information to the data type

```
data Expr a =
   -- literals
     Lit a
   -- integers
   | Neg (Expr Int)
   | Add (Expr Int) (Expr Int)
   | Mul (Expr Int) (Expr Int)
-- booleans
   | Not (Expr Bool)
   | Or  (Expr Bool) (Expr Bool)
   | And (Expr Bool) (Expr Bool)
-- mixed
   | Equ (Expr a) (Expr a)
   | If  (Expr Bool) (Expr a) (Expr a)
```

Did we do it???

Typed expressions

Limitation: return type of all constructors is **`Expr a`***!*

```
-- literals
Lit :: a -> Expr a
-- integers
Neg :: Expr Int -> Expr a
Add :: Expr Int -> Expr Int -> Expr a
Mul :: Expr Int -> Expr Int -> Expr a
-- booleans
Not :: Expr Bool -> Expr a
Or  :: Expr Bool -> Expr Bool -> Expr a
And :: Expr Bool -> Expr Bool -> Expr a
-- mixed
Equ :: Expr a -> Expr a -> Expr a
If  :: Expr Bool -> Expr a -> Expr a -> Expr a
```

statically ill-typed ☺

```
Add (Lit True) (Lit 4)
```

statically "well typed" ☹

```
Add (And (Lit True) (Lit False))
    (Lit 4)
```

Generalized algebraic data types

Allow you to specify the types of data constructors more precisely

```
data Expr a where
  -- literals
  Lit :: a -> Expr a
  -- integers
  Neg :: Expr Int -> Expr Int
  Add :: Expr Int -> Expr Int -> Expr Int
  Mul :: Expr Int -> Expr Int -> Expr Int
  -- booleans
Not :: Expr Bool -> Expr Bool
Or  :: Expr Bool -> Expr Bool -> Expr Bool
And :: Expr Bool -> Expr Bool -> Expr Bool
  -- mixed
Equ :: Eq a => Expr a -> Expr a -> Expr Bool
If  :: Expr Bool -> Expr a -> Expr a -> Expr a
```

can even define type class constraints and use this info in functions!

statically ill-typed ☺

```
Add (Lit True) (Lit 4)
```

statically ill-typed ☺

```
Add (And (Lit True) (Lit False))
    (Lit 4)
```