

Recursive Data Types

Meng Wang

with thanks to Koen Lindström Claessen

Modelling Arithmetic Expressions

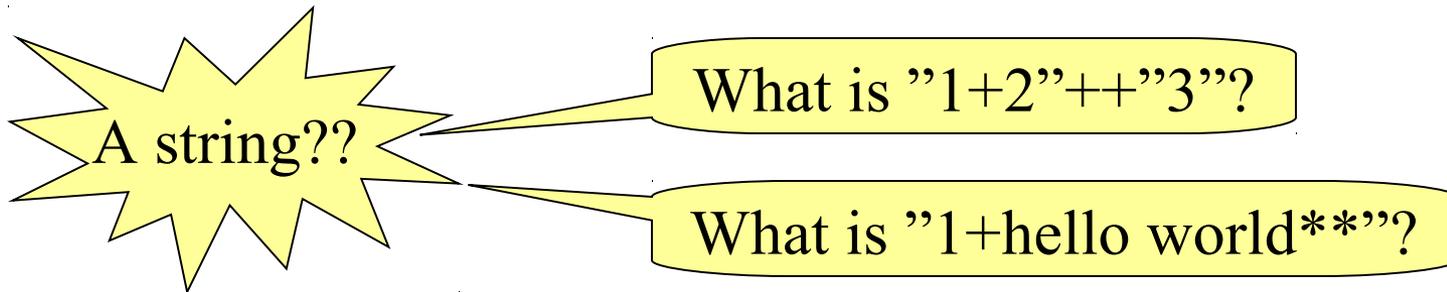
Imagine a program to help school-children learn arithmetic, which presents them with an expression to work out, and checks their answer.

What is $(1+2)*3$? **8**

Sorry, wrong answer!

Modelling Arithmetic Expressions

The expression $(1+2)*3$ is *data* as far as this program is concerned (**not** the same as 9!). How shall we represent it?



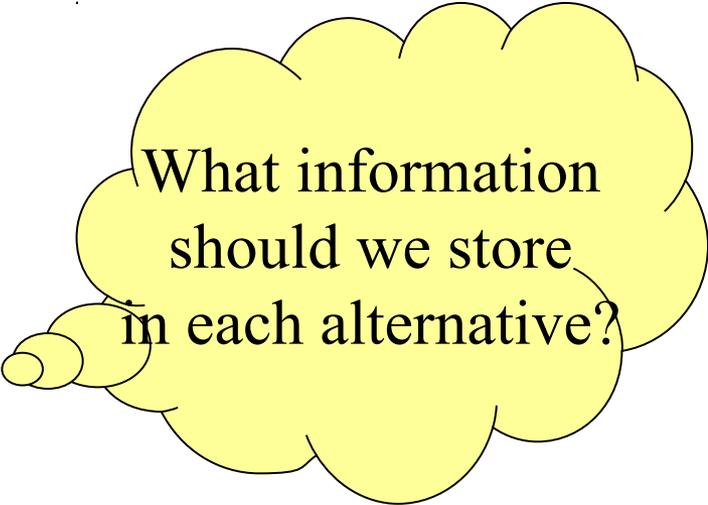
Modelling Expressions

Let's design a datatype to model *arithmetic expressions* -- not their values, but their structure.

An expression can be:

- a number n
- an addition $a+b$
- a multiplication $a*b$

```
data Expr =  
    Num  
    |  
    Add  
    |  
    Mul
```



What information
should we store
in each alternative?

A Difference

- There is a difference between
 - $17 \quad :: \text{Integer}$
 - $\text{Num } 17 \quad :: \text{Expr}$
- Why are these different?
 - Can do different things with them
 - Some things only work for one of them
 - So, their *types* should be different

Quiz

Can you define a function

$\text{eval} :: \text{Expr} \rightarrow \text{Integer}$

which *evaluates* an expression?

Example: $\text{eval} (\text{Add} (\text{Num } 1) (\text{Mul} (\text{Num } 2) (\text{Num } 3)))$

$\longrightarrow 7$

Hint: Recursive types often mean recursive functions!

Quiz

Can you define a function

$\text{eval} :: \text{Expr} \rightarrow \text{Integer}$

which *evaluates* an expression?

Use pattern matching: one equation for each case.

$\text{eval} (\text{Num } n) =$

$\text{eval} (\text{Add } a \ b) =$

$\text{eval} (\text{Mul } a \ b) =$

A and b are of
type Expr.

What can we put
here?

Quiz

Can you define a function

$\text{eval} :: \text{Expr} \rightarrow \text{Integer}$

which *evaluates* an expression?

$\text{eval} (\text{Num } n) = n$

$\text{eval} (\text{Add } a \ b) = \text{eval } a + \text{eval } b$

$\text{eval} (\text{Mul } a \ b) = \text{eval } a * \text{eval } b$

Recursive types mean
recursive functions!

Showing Expressions

Expressions will be more readable if we convert them to strings.

```
showExpr :: Expr -> String
```

```
showExpr (Num n)    = show n
```

```
showExpr (Add a b)  = showExpr a ++ "+" ++ showExpr b
```

```
showExpr (Mul a b)  = showExpr a ++ "*" ++ showExpr b
```

```
showExpr (Mul (Num 1) (Add (Num 2) (Num 3)))
```

→ "1*2+3"

Quiz

Which brackets are necessary?

$$1+(2+3)$$

$$1+(2*3)$$

$$1*(2+3)$$

What kind of expression *may* need to be bracketed?

When *does* it need to be bracketed?

Quiz

Which brackets are necessary?

$1+(2+3)$

NO!

$1+(2*3)$

NO!

$1*(2+3)$

YES!

What kind of expression *may* need to be bracketed?

Additions

When *does* it need to be bracketed?

Inside multiplications.

Idea

Format *factors* differently:

```
showExpr :: Expr -> String
showExpr (Num n)    = show n
showExpr (Add a b)  = showExpr a ++ "+" ++ showExpr b
showExpr (Mul a b)  = showFactor a ++ "*" ++ showFactor b
```

```
showFactor :: Expr -> String
showFactor (Add a b) = "(" ++ showExpr (Add a b) ++ ")"
showFactor e         = showExpr e
```

Making a Show instance

instance Show Expr **where**

show = showExpr

```
data Expr = Num Integer | Add Expr Expr | Mul Expr Expr  
deriving ( Show, Eq )
```

(Almost) Complete Program

New random seed

What's this?

An expression generator—needs to be written

```
questions :: IO ()
```

```
questions = do rnd <- newStdGen
```

```
  let e = unGen arbitrary rnd 10
```

```
  putStr ("What is "++show e++"? ")
```

```
  ans <- getLine
```

```
  putStrLn (if read ans == eval e
```

```
    then "Right!" else "Wrong!")
```

let: Give name to a result

Opposite of show

questions

Using QuickCheck Generators in Other Programs

- Test.QuickCheck exports
 - `unGen :: Gen a -> StdGen -> Int -> a`

QuickCheck
generator

Random
seed

Size parameter for
generation

- Size is used, for example, to bound Integers

Generating Arbitrary Expressions

```
instance Arbitrary Expr where  
  arbitrary = arbExpr
```

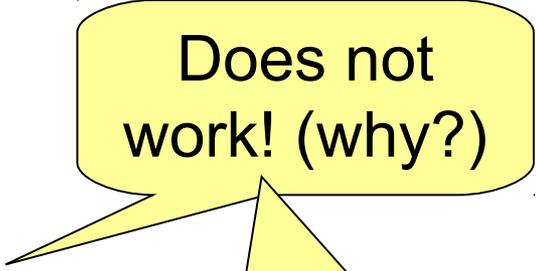
```
arbExpr :: Gen Expr
```

```
arbExpr =
```

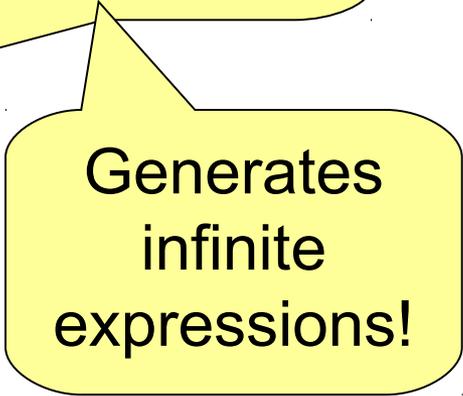
```
  oneof [ do n <- arbitrary  
          return (Num n)
```

```
    , do a <- arbExpr  
        b <- arbExpr  
        return (Add a b)
```

```
    , do a <- arbExpr  
        b <- arbExpr  
        return (Mul a b) ]
```



Does not
work! (why?)



Generates
infinite
expressions!

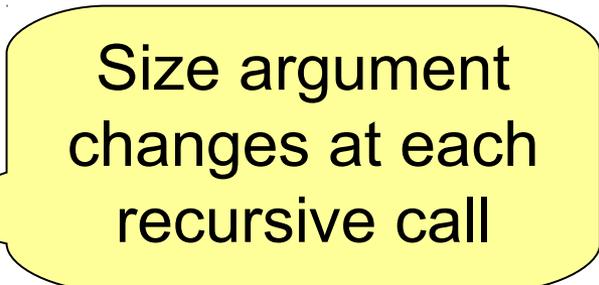
Generating Arbitrary Expressions

```
instance Arbitrary Expr where  
  arbitrary = sized arbExpr
```

```
arbExpr :: Int -> Gen Expr  
arbExpr s =  
  frequency [ (1, do n <- arbitrary  
              return (Num n))  
            , (s, do a <- arbExpr s'  
                  b <- arbExpr s'  
              return (Add a b))  
            , (s, do a <- arbExpr s'  
                  b <- arbExpr s'  
              return (Mul a b)) ]
```

```
where
```

```
s' = s `div` 2
```



Size argument
changes at each
recursive call

Demo

Main> questions

What is $-3*4*-1*-3*-1*-1$? -36

Right!

What is $15*4*(-2+-13+-14+13)$? -640

Wrong!

What is 0? 0

Right!

What is $(-4+13)*-9*13+7+15+12$? dunno

Program error: Prelude.read: no parse

The Program

failing

```
putStrLn (if read ans==eval e  
            then "Right!" else "Wrong!")
```

cannot fail

```
putStrLn (if ans==show (eval e)  
            then "Right!" else "Wrong!")
```

Reading Expressions

- How about a function
 - `readExpr :: String -> Expr`

- Such that

- `readExpr "12+173" =`
 - `Add (Num 12) (Num 173)`

- `readExpr "12+3*4" =`
 - `Add (Num 12) (Mul (Num 3) (Num 4))`



We see how to
implement this
in the next
lecture

Symbolic Expressions

- How about expressions with variables in them?

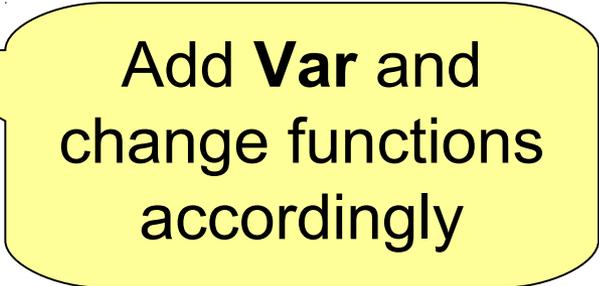
```
data Expr = Num Integer
```

```
    | Add Expr Expr
```

```
    | Mul Expr Expr
```

```
    | Var Name
```

```
type Name = String
```



Add **Var** and
change functions
accordingly

Gathering Variables

It is often handy to know exactly which variables occur in a given expression

$\text{vars} :: \text{Expr} \rightarrow [\text{Name}]$

$\text{vars} = ?$

Gathering Variables

It is often handy to know exactly which variables occur in a given expression

$\text{vars} :: \text{Expr} \rightarrow [\text{Name}]$

$\text{vars} (\text{Num } n) = []$

$\text{vars} (\text{Add } a \ b) = \text{vars } a \ \text{'union'} \ \text{vars } b$

$\text{vars} (\text{Mul } a \ b) = \text{vars } a \ \text{'union'} \ \text{vars } b$

$\text{vars} (\text{Var } x) = [x]$



From Data.List;
combines two
lists without
duplication

Table of values
for variables

ting Expressions

We would like to evaluate expressions with variables. What is the type?

$\text{eval} :: [(\text{Name}, \text{Integer})] \rightarrow \text{Expr} \rightarrow \text{Integer}$

$\text{eval env (Num } n) = n$

$\text{eval env (Var } y) = \text{fromJust (lookup } y \text{ env)}$

$\text{eval env (Add } a \text{ } b) = \text{eval env } a + \text{eval env } b$

$\text{eval env (Mul } a \text{ } b) = \text{eval env } a * \text{eval env } b$

Variable to
differentiate wrt.

ic Differentiation

Differentiating an expression produces a new expression. We implement it as:

$\text{diff} :: \text{Expr} \rightarrow \text{Name} \rightarrow \text{Expr}$

$\text{diff} (\text{Num } n) \ x = \text{Num } 0$

$\text{diff} (\text{Var } y) \ x \mid x == y = \text{Num } 1$

$\mid x \neq y = \text{Num } 0$

$\text{diff} (\text{Add } a \ b) \ x = \text{Add} (\text{diff } a \ x) (\text{diff } b \ x)$

$\text{diff} (\text{Mul } a \ b) \ x = \text{Add} (\text{Mul } a (\text{diff } b \ x)) (\text{Mul } b (\text{diff } a \ x))$

Testing differentiate

```
Main> diff (Mul (Num 2) (Var "x")) "x"  
2*1+0*x
```

Not quite what we expected!
-- not *simplified*

What happens?

$$\frac{d}{dx}(2*x) = 2$$

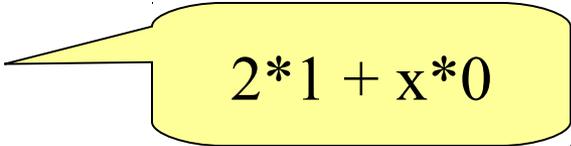
differentiate (Mul (Num 2) (Var "x")) "x"

→ Add (Mul (Num 2) (differentiate (Var "x") "x"))

(Mul (Var "x") (differentiate (Num 2) "x"))

→ Add (Mul (Num 2) (Num 1))

(Mul (Var "x") (Num 0))


$$2*1 + x*0$$

How can we make differentiate simplify the result?

“Smart” Constructors

- Define

```
add :: Expr -> Expr -> Expr
add (Num 0) b          = b
add a      (Num 0)    = a
add (Num x) (Num y)  = Num (x+y)
add a      b          = Add a b
```

more
simplification
is possible...

By using **add** instead of **Add**,
certain simplifications are
performed when constructing
the expression!

Testing add

```
Main> Add (Num 2) (Num 5)
```

```
2+5
```

```
Main> add (Num 2) (Num 5)
```

```
7
```

Symbolic Differentiation

Differentiating an expression produces a new expression. We implement it as:

$\text{diff} :: \text{Expr} \rightarrow \text{Name} \rightarrow \text{Expr}$

$\text{diff} (\text{Num } n) \ x = \text{Num } 0$

$\text{diff} (\text{Var } y) \ x \mid x == y = \text{Num } 1$

$\mid x \neq y = \text{Num } 0$

$\text{diff} (\text{Add } a \ b) \ x = \text{add} (\text{diff } a \ x) (\text{diff } b \ x)$

$\text{diff} (\text{Mul } a \ b) \ x = \text{add} (\text{mul } a (\text{diff } b \ x)) (\text{mul } b (\text{diff } a \ x))$

note

note

note

“Smart” Constructors -- mul

- How to define mul?

```
mul :: Expr -> Expr -> Expr
mul (Num 0) b          = Num 0
mul a      (Num 0)    = Num 0
mul (Num 1) b         = b
mul a      (Num 1)    = a
mul (Num x) (Num y)  = Num (x*y)
mul a      b          = Mul a b
```

Expressions

- Expr as a datatype can represent expressions
 - Unsimplified
 - Simplified
 - Results
 - Data presented to the user
- Need to be able to convert between these

An Expression Simplifier

- Simplification function
 - `simplify :: Expr -> Expr`

```
simplify :: Expr -> Expr  
simplify e | null (vars e) = ?  
...
```

You continue at the group
exercises!

Testing the Simplifier

```
arbExpr :: Int -> Gen Expr
```

```
arbExpr s =
```

```
  frequency [ (1, do n <- arbitrary
               return (Num n))
             , (s, do a <- arbExpr s'
               b <- arbExpr s'
               return (Add a b))
             , (s, do a <- arbExpr s'
               b <- arbExpr s'
               return (Mul a b))
             , (1, do x <- elements [”x”,”y”,”z”]
               return (Var x))]
```

where

```
s' = s `div` 2
```

Testing an Expression Simplifier

- (1) Simplification should not change the value

```
prop_SimplifyCorrect e env =  
  eval env e == eval env (simplify e)
```

Generate lists of
values *for variables*

```
prop_SimplifyCorrect e (Env env) =  
  eval env e == eval env (simplify e)
```

Testing an Expression Simplifier

```
data Env = Env [(Name,Integer)]  
  deriving ( Eq, Show )  
  
instance Arbitrary Env where  
  arbitrary =  
    do a <- arbitrary  
      b <- arbitrary  
      c <- arbitrary  
      return (Env [("x",a),("y",b),("z",c)])
```

Testing an Expression Simplifier

- (2) Simplification should do a good job

```
prop_SimplifyNoJunk e =  
  noJunk (simplify e)  
where  
  noJunk (Add a b) = not (isNum a && isNum b)  
                    && noJunk a && noJunk b  
  
  ...
```

You continue at the group
exercises!

Forthcoming Group Exercise

- Build and test an expression simplifier!
- I found *many subtle bugs* in my own simplifier!
 - Often simplifier goes into an infinite loop

Summary

- Recursive data-types can take many forms other than lists
- Recursive data-types can model *languages* (expressions, natural languages, programming languages)
- Functions working with recursive types are often recursive themselves
- When generating random elements in recursive datatypes, think about the *size*

Next Time

- How to write *parsers*
 - `readExpr :: String -> Expr`
- Case study: example of other recursive datatype
 - a simple game: "the zoo"
 - guessing animals using yes/no questions