

## Recursive Data Types

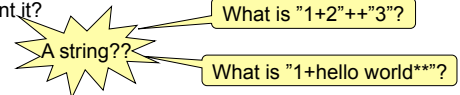


## Modelling Arithmetic Expressions

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!

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?

## 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 Integer
  | Add Expr Expr
  | Mul Expr Expr
```

A recursive data type

## Examples

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

The expression: is represented by:

2	Num 2
2+2	Add (Num 2) (Num 2)
$(1+2)*3$	Mul (Add (Num 1) (Num 2)) (Num 3)
$1+2*3$	Add (Num 1) (Mul (Num 2) (Num 3))

## Example: Evaluation

Define a function

```
eval :: Expr -> Integer
```

which *evaluates* an expression?

*Example:* `eval (Add (Num 1) (Mul (Num 2) (Num 3)))`  
evaluates to 7

*Hint:* Recursive types often mean recursive functions!

```
eval :: Expr -> Integer
eval (Num n) = n
eval (Add a b) = eval a + eval b
eval (Mul a b) = eval a * eval b
```

Use pattern matching: one equation for each case.

a and b are of type Expr.

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
```

Main> 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?  
When *does* it need to be bracketed?

Additions  
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

```

questions :: IO ( )
questions =
  do rnd <- newStdGen
  let e = unGen arbitrary rnd 3
  putStr ("What is "++show e++"? ")
  ans <- getLine
  putStrLn (if read ans==eval e
            then "Right!" else "Wrong!")
  questions

```

Annotations:

- New random seed
- An expression generator—needs to be written
- What's this?
- let: Give name to a result
- Opposite of show

### Using QuickCheck Generators in Other Programs

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

Annotations:

- QuickCheck generator
- Random seed
- Size parameter for generation

- Size is used, for example, to bound Integers, size of data structures etc.

### 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) ]

```

Annotations:

- 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

```

Annotation: 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

Crucial line:

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

failing

```
ans == show (eval e)
```

cannot fail

## 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

```
vars :: Expr -> [Name]
```

```
vars (Num n) = []
```

```
vars (Add a b) = vars a `union` vars b
```

```
vars (Mul a b) = vars a `union` vars b
```

```
vars (Var x) = [x]
```

From Data.List;  
combines two  
lists without  
duplication

## Evaluating Expressions

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

Table of values for variables

```
eval :: [(Name, Integer)] -> Expr -> Integer
eval env (Num n) = n
eval env (Var y) = fromJust (lookup y env)
eval env (Add a b) = eval env a + eval env b
eval env (Mul a b) = eval env a * eval env b
```

Prelude> :i lookup

lookup :: (Eq a) => a -> [(a, b)] -> Maybe b

## Symbolic Differentiation

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

Variable to differentiate wrt.

```
diff :: Expr -> Name -> Expr
diff (Num n) x = Num 0
diff (Var y) x | x==y = Num 1
                | x/=y = Num 0
diff (Add a b) x = Add (diff a x) (diff b x)
diff (Mul a b) x = Add (Mul a (diff b x))
                      (Mul b (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

```
diff :: Expr -> Name -> Expr
diff (Num n) x = Num 0
diff (Var y) x
  | x==y = Num 1
  | x/=y = Num 0
diff (Add a b) x = add (diff a x) (diff b x)
diff (Mul a b) x = add (mul a (diff b x))
                      (mul b (diff a x))
```

## “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) = ?
...
```

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
```

Cut'n'paste here should be refactored

## Testing an Expression Simplifier

- (1) Simplification should not change the value

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

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

Generate lists of values for variables

## 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 exercises!

## Exercises

- Build and test an expression simplifier!
- I found *many subtle bugs* in my own simplifier!
  - Often simplifier goes into an infinite loop
  - Use `verboseCheck` instead of `quickCheck` (prints test case *before* every test, so you see them even if the test loops or crashes)

## 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`