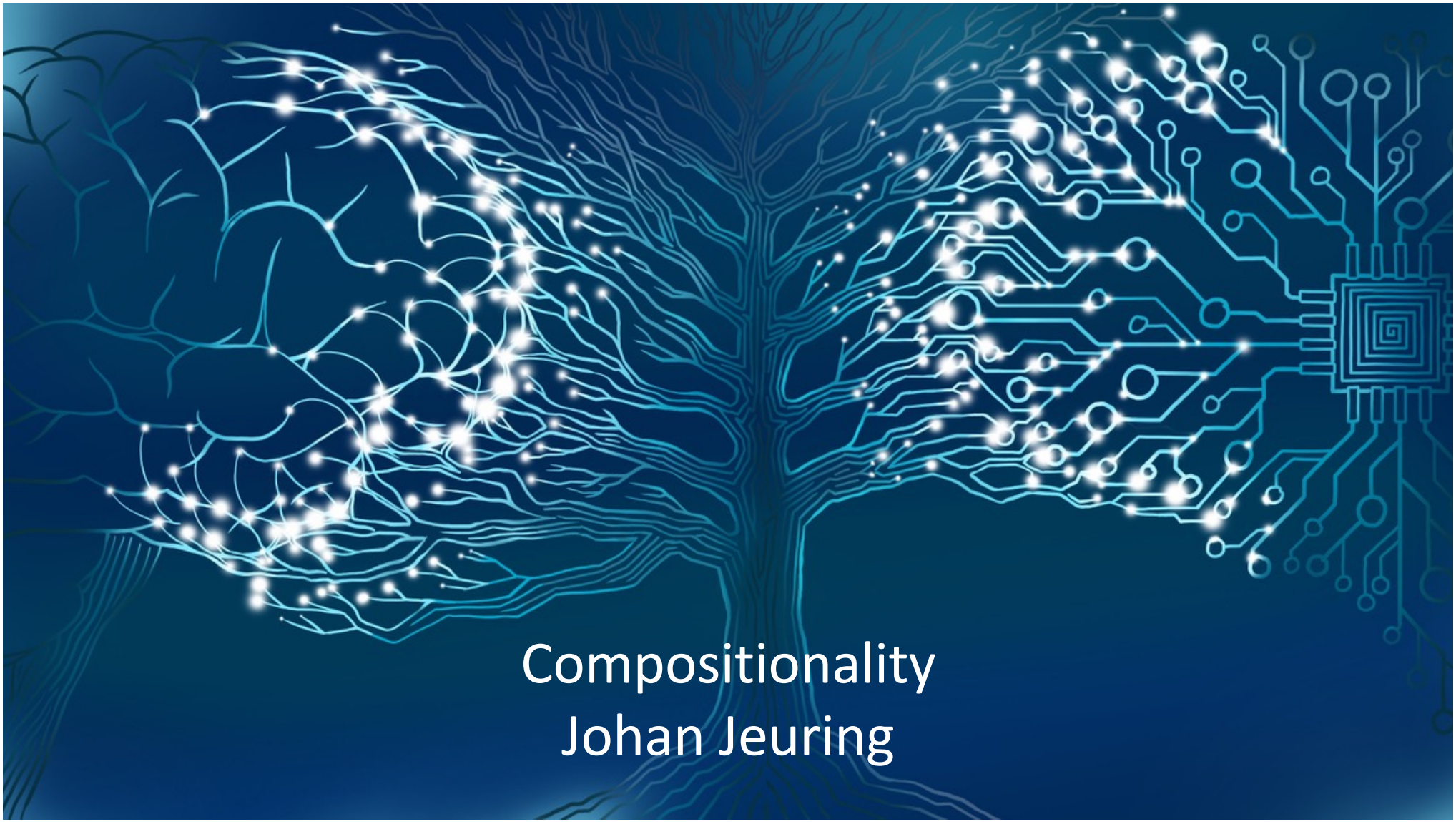
Compositionality
Johan Jeuring

Utrecht University

Last week

Parser combinators …

… designing grammars and parsers

Involving semantic functions

Utrecht University

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
EIPRNS

Enable answers by SMS

Utrecht University

About the labs and autograding

From now on, we will merge the group from BBG 165 (today KBG 208) with the groups BBG 223 and BBG 209

The first feedback moment for the iCal lab was on Sunday; the second is on Wednesday

We will try to get the autograder results back to you today

Note the autograder doesn't answer particular questions: please asks them in the labs. TAs are happy to help!

Utrecht University

Compositionality

Lecture notes: 5.1-3

Utrecht University

Compiler phases

- Lexing and parsing
- Analysis and type checking
- Desugaring
- Optimization
- Code generation

Not all compilers implement all phases
Some compilers implement many more phases

Abstract syntax trees

Abstract syntax trees (ASTs) play a central role:

- Some phases build ASTs (parsing)
- Most phases traverse ASTs (analysis, type checking, code generation)
- Some phases traverse one AST and build another (such as desugaring)

Learning goals

Parsers build abstract syntax trees

Learning goals:

- Develop software to traverse ASTs systematically to compute all sorts of information
- Write **compositional** functions, also known as folds, on (possibly mutually recursive) datatypes
- Explain the relation between concrete syntax, abstract syntax, datatypes, algebras and folds.

Utrecht University

Example: Matched parentheses

S -> (S)S | ε

Examples: (())(()), (), (()()), …

Abstract syntax:

```
data Parens = Match Parens Parens
            | Empty
```

Utrecht University

Counting matching pairs of parentheses

```
data Parens = Match Parens Parens
            | Empty

runParser (count <$> parens) “(())(())”  ⇒  4

count :: Parens          → Int
count    (Match p1 p2) = (count p1 + 1) + count p2
count    Empty          = 0
```

The definition of `count` mirrors the recursive structure of the datatype

Utrecht University

More semantic functions

```
data Parens = Match Parens Parens
            | Empty

runParser (depth <$> parens) “(())(())”   ⇒   2

depth :: Parens          → Int
depth    (Match p1 p2) = (depth p1 + 1) ‘max‘ depth p2
depth    Empty          = 0

runParser (print <$> parens) “(())(())”   ⇒   “(())(())”

print :: Parens          → String
print    (Match p1 p2) = "(" ++ print p1 ++ ")" ++ print p2
print    Empty          = ""
```

Utrecht University

Capturing the recursive structure I

All functions have the following structure:

```
f :: Parens          → …
f    (Match p1 p2) = … (f p1) … (f p2) …
f    Empty         = …
```

We abstract from this structure

Utrecht University

Capturing the recursive structure II

```
f :: Parens          → …
f    (Match p1 p2) = … (f p1) … (f p2) …
f    Empty         = …
```

Utrecht University

Capturing the recursive structure II

```
f :: Parens          → r
f    (Match p1 p2) = match (f p1) (f p2)
f    Empty         = empty


Match :: Parens → Parens → Parens
match :: r      → r      → r

Empty :: Parens
empty :: r
```

Utrecht University

Algebra and fold

Each semantic function needs a different instance of `match`, `empty` and `r`

```
type ParensAlgebra r = (r → r → r -- match
                       ,r)        -- empty

foldParens :: ParensAlgebra r → Parens → r
foldParens (match,empty) = f
  where f (Match p1 p2) = match (f p1) (f p2)
        f Empty         = empty
```

Utrecht University

count revisited

```
type ParensAlgebra r = (r → r → r -- match
                       ,r)        -- empty

foldParens :: ParensAlgebra r → Parens → r
foldParens (match,empty) = f
  where f (Match p1 p2) = match (f p1) (f p2)
        f Empty          = empty

count :: Parens         → Int
count    (Match p1 p2) = (count p1 + 1) + count p2
count    Empty          = 0

countAlgebra = (\l r → l+1+r, 0)

count = foldParens countAlgebra
```

Utrecht University

depth and print

```
depthAlgebra = (\l r → max (l+1) r, 0)
depth        = foldParens depthAlgebra

printAlgebra = (\l r → "(”++l++")”++r, “”)
print        = foldParens printAlgebra
```

Utrecht University

Example: Expressions

Examples: 2+(-3+4), -(2+8)+16

E → E + E
E → -E
E → Nat
E → (E)

Associative operator grammar transformation

E → E' + E
E' → -E'
E' → Nat
E' → (E)

Utrecht University

Abstract syntax

Based on original grammar:

```
data E = Add E E
       | Neg E
       | Num Int
```

Utrecht University

Functions on expressions

```
data E = Add E E
       | Neg E
       | Num Int

eval :: E            → Int
eval    (Add e1 e2) = eval e1 + eval e2
eval    (Neg e)     = - (eval e)
eval    (Num i)     = i
```

The structure of the function again reflects the structure of the datatype

Utrecht University

An expression algebra

```
data E = Add E E
       | Neg E
       | Num Int

Add :: E → E → E
Neg :: E → E
Num :: Int → E

type Ealgebra r = (r → r → r -- add
                  ,r → r     -- neg
                  ,Int → r)  -- num
```

Utrecht University

Folds on expressions

```
type Ealgebra r = (r → r → r -- add
                  ,r → r     -- neg
                  ,Int → r)  -- num

foldE :: EAlgebra r → E → r
foldE (add,neg,num) = f
  where f (Add e1 e2) = add (f e1) (f e2)
        f (Neg e)     = neg (f e)
        f (Num n)     = num n

evalAlgebra :: EAlgebra Int
evalAlgebra =  ((+),negate,id)

eval        =  foldE evalAlgebra
```

Q2

Folds for all data types

Utrecht University

Trees

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)

Leaf :: a → Tree a
Node :: Tree a → Tree a → Tree a

type TreeAlgebra a r = (a → r       -- leaf
                       ,r → r → r) -- node

foldTree :: TreeAlgebra a r → Tree a → r
foldTree (leaf,node) = f
  where f (Leaf x)   = leaf x
        f (Node l r) = node (f l) (f r)
```

Utrecht University

Tree algebra examples

```
sizeAlgebra    :: TreeAlgebra a Int
sumAlgebra     :: TreeAlgebra Int Int
inorderAlgebra :: TreeAlgebra a [a]
reverseAlgebra :: TreeAlgebra a (Tree a)

sizeAlgebra    =  (const 1,(+))
sumAlgebra     =  (id,(+))
inorderAlgebra =  ((:[]),++)
reverseAlgebra =  (Leaf,flip Node)

idAlgebra      :: TreeAlgebra a (Tree a)
idAlgebra      =  (Leaf,Node)
```

Utrecht University

User-defined lists

```
data List a = Nil
            | Cons a (List a)

Nil  :: List a
Cons :: a → List a → List a

type ListAlgebra a r = (r
                       ,a → r → r)

foldList :: ListAlgebra a r → List a → r
foldList (nil,cons) = f
  where f Nil          = nil
        f (Cons x xs) = cons x (f xs)
```

Utrecht University

Built-in lists

```
data [a] = []
         | a : [a]

[]  :: [a]
(:) :: a → [a] → [a]

type LAlgebra a r = (r
                    ,a → r → r)

foldL :: LAlgebra a r →[a] →r
foldL (nil,cons) = f
  where f []     = nil
        f (x:xs) = cons x (f xs)
```

Utrecht University

foldL versus foldr

```
type LAlgebra a r = (r
                    ,a → r → r)

foldL :: LAlgebra a r →[a] →r
foldL (nil,cons) = f
  where f []     = nil
        f (x:xs) = cons x (f xs)

foldr :: (a → r → r) → r → [a] → r
foldr cons nil []     = nil
foldr cons nil (x:xs) = cons x (foldr cons nil xs)

foldr cons nil == foldL (nil,cons)
```

Utrecht University

Maybe

Non-recursive datatypes:

```
data Maybe a = Nothing
             | Just a

Nothing :: Maybe a
Just    :: a → Maybe a

type MaybeAlgebra a r = (r,a →r)

foldMaybe :: MaybeAlgebra a r → Maybe a → r
foldMaybe (nothing,just) = f
  where f Nothing  = nothing
        f (Just x) = just x
```

Utrecht University

foldMaybe versus maybe

```
type MaybeAlgebra a r = (r,a →r)

foldMaybe :: MaybeAlgebra a r → Maybe a → r
foldMaybe (nothing,just) = f
  where f Nothing  = nothing
        f (Just x) = just x

maybe :: r → (a → r) → Maybe a → r
maybe nothing just Nothing  = nothing
maybe nothing just (Just x) = just x

maybe nothing just == foldMaybe (nothing,just)
```

Utrecht University

Bool

```
data Bool = True
          | False

True  :: Bool
False :: Bool

type BoolAlgebra r = (r,r)

foldBool :: BoolAlgebra r → Bool → r
foldBool (true,false) True  = true
foldBool (true,false) False = false
```

Utrecht University

foldBool versus if-then-else

```
type BoolAlgebra r = (r,r)

foldBool :: BoolAlgebra r → Bool → r
foldBool (true,false) True  = true
foldBool (true,false) False = false


foldBool (true,false) x == if x then true else false
```

Q3,4

Utrecht University

Summary

For a datatype T, we can define a fold function as follows:

- Define an algebra type TAlgebra that is parameterized over all of T's parameters, plus a result type r
- The algebra is a tuple containing one component per constructor function
- The types of the components are like the types of the constructor functions, but all occurrences of T are replaced with r
- The fold function is defined by traversing the data structure, replacing constructors with their corresponding algebra components, and recursing where required

Advantages of using folds

- A systematic recursion “design” pattern that is well known and easy to understand (compare with iterator)
- Using a fold forces us to define semantics in a compositional fashion – the semantics of a whole term is composed from the semantics of its subterms
- The systematic nature of a fold makes it easy to combine several folds into one (fusion). This is essential for efficiency in a compiler
- Compilers cannot determine the recursive structure of `general recursive’ functions