Session 5

Defining Types

Universiteit Antwerpen

Data Declarations

A new type can be defined by specifying its set of values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

Universiteit Antwerpen

Values of new types can be used in the same ways as those of built-in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers        :: [Answer]
answers         = [Yes,No,Unknown]

flip           :: Answer → Answer
flip Yes        = No
flip No         = Yes
flip Unknown    = Unknown
```

Functie zonder argumenten die een lijst van Answer data teruggeeft

Def. Vd functie: geef constante lijst van alle 3 de antwoorden terug Indien gewenst zou je er dubbelen in kunnen steken maar dat is hier natuurlijk ongewenst...

2de functie die het Answer type gebruikt op een meer zinvolle manier

Universiteit Antwerpen

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

Ctor pars

we can define:

```
square          :: Float → Shape
square n         = Rect n n

area            :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Ctor call met argn

Universiteit Antwerpen

Similarly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

Ctor met 1 parameter

we can define:

```
return          :: a → Maybe a
return x         = Just x

(>>=) :: Maybe a → (a → Maybe b) → Maybe b
Nothing >>= _  = Nothing
Just x  >>= f = f x
```

Universiteit Antwerpen

Recursive Types

In Haskell, new types can be defined in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat $\rightarrow$ Nat.

Universiteit Antwerpen

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function (1 +).

- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))  =  3
```

Universiteit Antwerpen

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

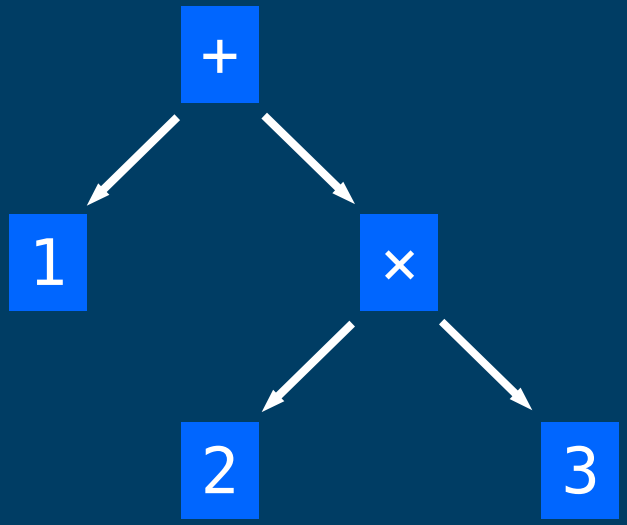
```
nat2int           :: Nat → Int
nat2int Zero       = 0
nat2int (Succ n) = 1 + nat2int n


int2nat           :: Int → Nat
int2nat 0          = Zero
int2nat (n+1)      = Succ (int2nat n)
```

Universiteit Antwerpen

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



Using recursion, a suitable new type to represent such expressions can be defined by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Universiteit Antwerpen

Using recursion, it is now easy to define functions that process expressions. For example:

```
size          :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval          :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Universiteit Antwerpen

Exercises

(1) Define an add function, once using conversion functions (int2nat, ...) and once using recursion.

(2) Define a type BinaryTree, with nodes and leafs (nodes have 2 subtrees, leafs don't). Each node/leaf has a value. Define a function “occurs”, which checks if a given value occurs in a tree.

Universiteit Antwerpen