# Data structures

The INFDEV@HR Team

Hogeschool Rotterdam
Rotterdam, Netherlands

# Introduction

# Introduction



## Lecture topics

- Let
- Tuples
- Discriminated unions (polymorphism)

# Let-in

# Let-in



## Idea

- Sometimes we wish to give a name to a value or a computation, to reuse later
- This construct is called `let-in`
- We could then say something like `let age = 9 in age + age`
- We can nest `let-in` constructs, and then say something like `let age = 9 in (let x = 2 in age * x)`

# Let-in



## Idea

- Sometimes we wish to give a name to a value or a computation, to reuse later
- This construct is called `let-in`
- We could then say something like `let age = 9 in age + age`
- We can nest `let-in` constructs, and then say something like `let age = 9 in (let x = 2 in age * x)`
- This makes code significantly more readable, as it looks like a series of declarations top-to-bottom

# Let-in



## Idea

- Lets are simply translated to function applications
- `let x = t in u` simply becomes `(`$\lambda$`x`$\rightarrow$` u) t`

# Let-in



```
let age = 9 in (age + age)
```

# Let-in



```
let age = 9 in (age + age)
```

```
let age = 9 in (age + age)
```

# Let-in



```
let age = 9 in (age + age)
```

# Let-in



```
let age = 9 in (age + age)
```

$((\lambda \text{age} \to (\text{age} + \text{age}))\ 9)$

# Let-in



```
((λage→(age + age)) 9)
```

# Let-in



```
((λage→(age + age)) 9)
```

```
((λage→(age + age)) 9)
```

# Let-in

```
((λage→(age + age)) 9)
```

# Let-in



$((\lambda\texttt{age} \rightarrow (\texttt{age} + \texttt{age}))\ 9)$

$(9 + 9)$

# Let-in



```
(9 + 9)
```

# Let-in



```
(9 + 9)
```

```
(9 + 9)
```

# Let-in



```
(9 + 9)
```

# Let-in



```
(9 + 9)
```

```
18
```

# Data types

# Data types



## Overview

- We now move on to ways to define data types
- The definitions will be both **minimal** and **composable**
- Classes, polymorphism, etc. can all be rendered under our definitions, so we miss nothing substantial

# Data types

## Overview

Notice: from now on we will start ignoring the reduction steps for simple terms such as 3+3, `x = 0`, etc. for brevity

# Data types

## Minimality

- The lambda calculus has so far proven very powerful, despite its size
- We do not need hundreds of different operators, we can simply build them[a]
- The only extension needed is purely syntactic in nature to make it more mnemonic, but this is only skin-deep and requires no change to the underlying mechanisms of the lambda calculus

[a] and more

# Data types



## Minimality

- In defining data types we wish to maintain this minimality
- We do not want dozens of separate, competing data types all slightly overlapping

## Fundamental scenarios

- **Tuples**: storing multiple things together at the same time, like the fields and methods in a class
- **Unions**: storing either one of various things at a time, like an interface that is exactly one of its concrete implementors

### The importance of composition

- We just need to cover the case of two items, higher numbers come through composition
- For example, given the ability to store a pair, we can build a pair of pairs to create arbitrary tuples
- Similarly, given the ability to store either of two values, we can build either of many values with nesting

# Tuples

# Tuples



- A pair of values is defined simply as something that stores these two values
- We can extract them by giving the pair a function that will receive the values

$$(\lambda x\ \ y \rightarrow\ (\lambda f \rightarrow ((f\ \ x)\ \ y)))$$

# Tuples



```
(1 , 2)
```

# Tuples



```
(1, 2)
```

```
(((,) 1) 2)
```

# Tuples



```
(( (,)  1)  2)
```

# Tuples



```
(( (,)  1)  2)
```

```
(( (λx y→ (λf→((f x) y)))  1)  2)
```

# Tuples



$$\boxed{(((\lambda \texttt{x}\ \ \texttt{y} \rightarrow\ (\lambda \texttt{f} \rightarrow ((\texttt{f}\ \ \texttt{x})\ \ \texttt{y})))\ \ 1)\ \ 2)}$$

# Tuples



$$(((\lambda \mathtt{x}\ \mathtt{y} \rightarrow (\lambda \mathtt{f} \rightarrow ((\mathtt{f}\ \mathtt{x})\ \mathtt{y})))\ 1)\ 2)$$

$$(\boxed{((\lambda \mathtt{x}\ \mathtt{y} \rightarrow (\lambda \mathtt{f} \rightarrow ((\mathtt{f}\ \mathtt{x})\ \mathtt{y})))\ 1)}\ 2)$$

# Tuples

$$( \; ((\lambda x \; y \rightarrow \; (\lambda f \rightarrow ((f \;\; x) \;\; y))) \;\; 1) \;\;\; 2)$$

# Tuples

$$(\ ((\lambda x\ y \rightarrow\ (\lambda f \rightarrow ((f\ x)\ y)))\ 1)\ \ 2)$$

$$((\lambda y\ \ f \rightarrow ((f\ \ 1\ )\ \ y))\ \ 2)$$

# Tuples



$$((\lambda \mathtt{y}\ \ \mathtt{f} \rightarrow ((\mathtt{f}\ \ 1)\ \ \mathtt{y}))\ \ 2)$$

# Tuples



```
((λy f→((f 1) y)) 2)
```

```
((λy f→((f 1) y)) 2)
```

# Tuples

$((\lambda \texttt{y f} \rightarrow ((\texttt{f 1) y)) 2)$

# Tuples

$((\lambda \mathtt{y}\ \mathtt{f} \rightarrow ((\mathtt{f}\ 1)\ \mathtt{y}))\ 2)$

$(\lambda \mathtt{f} \rightarrow ((\mathtt{f}\ 1)\ 2))$

# Tuples



- We can define two utility functions that, given a pair, extract the first or second value
- They are usually called $\pi_1$ and $\pi_2$, or `fst` and `snd`

```
(λp→(p (λx y→x)))
```

```
(λp→(p (λx y→y)))
```

# Tuples

```
(π₁ (1, 2))
```

# Tuples

$(\pi_1\ (1,\ 2))$

$(\boxed{\pi_1}\ (1,\ 2))$

# Tuples

$(\boxed{\pi_1}\ (1,\ 2))$

HOGESCHOOL ROTTERDAM

# Tuples



$(\ \pi_1\ (1,\ 2))$

$(\ (\lambda p \rightarrow (p\ (\lambda x\ y \rightarrow x)))\ (1,\ 2))$

# Tuples



```
((λp→(p (λx y→x))) (1, 2))
```

# Tuples



```
((λp→(p (λx y→x))) (1, 2))
```

```
((λp→(p (λx y→x))) (((,) 1) 2))
```

# Tuples



$((\lambda p \rightarrow (p\ (\lambda x\ y \rightarrow x)))\ ((\,\boxed{(,)}\ 1)\ 2))$

# Tuples

```
((λp→(p (λx y→x))) (( (,) 1) 2))
```

```
((λp→(p (λx y→x))) (( (λx y→ (λf→((f x) y))) 1)
     2))
```

# Tuples



```
((λp→(p (λx y→x))) (((λx y→ (λf→((f x) y)))
    1) 2))
```

# Tuples



```
((λp→(p (λx y→x))) ((((λx y→ (λf→((f x) y)))
    1) 2))
```

```
((λp→(p (λx y→x))) (
    ((λx y→ (λf→((f x) y))) 1) 2))
```

# Tuples

```
((λp→(p (λx y→x))) (
    ((λx y→ (λf→((f x) y))) 1) 2))
```

# Tuples

```
((λp→(p (λx y→x))) (
    ((λx y→ (λf→((f x) y))) 1) 2))
```

```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

# Tuples

```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

# Tuples

```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

# Tuples



```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

# Tuples



```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

```
((λp→(p (λx y→x))) (λf→((f 1) 2)))
```

# Tuples

$$\boxed{\texttt{((}\lambda\texttt{p}\rightarrow\texttt{(p (}\lambda\texttt{x y}\rightarrow\texttt{x))) (}\lambda\texttt{f}\rightarrow\texttt{((f 1) 2)))}}$$

# Tuples



```
((λp→(p (λx y→x))) (λf→((f 1) 2)))
```

```
((λp→(p (λx y→x))) (λf→((f 1) 2)))
```

# Tuples



$$((\lambda \mathtt{p} \rightarrow (\mathtt{p}\ (\lambda \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{x})))\ (\lambda \mathtt{f} \rightarrow ((\mathtt{f}\ 1)\ 2)))$$

# Tuples



$((\lambda p \rightarrow (p\ (\lambda x\ y \rightarrow x)))\ (\lambda f \rightarrow ((f\ 1)\ 2)))$

$(\ (\lambda f \rightarrow ((f\ 1)\ 2))\ \ (\lambda x\ \ y \rightarrow x))$

# Tuples



```
((λf→((f 1) 2)) (λx y→x))
```

# Tuples



$((\lambda f \rightarrow ((f\ \ 1)\ \ 2))\ \ (\lambda x\ \ y \rightarrow x))$

$((\lambda f \rightarrow ((f\ \ 1)\ \ 2))\ \ (\lambda x\ \ y \rightarrow x))$

# Tuples



$$((\lambda f \rightarrow ((f\ 1)\ 2))\ (\lambda x\ y \rightarrow x))$$

# Tuples



$((\lambda f \to ((f\ 1)\ 2))\ (\lambda x\ y \to x))$

$(((\lambda x\ y \to x)\ 1)\ 2)$

# Tuples



```
(((λx y→x) 1) 2)
```

# Tuples



```
(((λx y→x) 1) 2)
```

```
(((λx y→x) 1) 2)
```

# Tuples



$$( \boxed{((\lambda x\ y \rightarrow x)\ 1)}\ \ 2)$$

# Tuples



$$( \boxed{((\lambda x\ y \rightarrow x)\ 1)}\ \ 2)$$

$$((\lambda y \rightarrow \boxed{1})\ \ 2)$$

# Tuples



```
((λy→1) 2)
```

# Tuples

```
((λy→1) 2)
```

```
((λy→1) 2)
```

# Tuples



```
((λy→1) 2)
```

# Tuples

```
((λy→1) 2)
```

```
1
```

# Tuples

## Pair of values

We should expect that $\pi_1$ and $\pi_2$ are inverse operations to constructing a pair, as they destroy it

# Tuples



$\texttt{let p = (1, 2) in ((}\pi_1\texttt{ p), (}\pi_2\texttt{ p))}$

# Tuples



```
let p = (1, 2) in ((π1 p), (π2 p))
```

```
((π1 (1, 2)), (π2 (1, 2)))
```

# Tuples



$$((\pi_1 \; \texttt{(1, 2)}), (\pi_2 \; \texttt{(1, 2)}))$$

# Tuples

```
((π₁ (1, 2)), (π₂ (1, 2)))
```

```
((π₁ (1, 2)), (π₂ (1, 2)))
```

# Tuples

Data structures

The INFDEV@HR Team

Introduction

Let-in

Data types

Tuples

Discriminated unions

Conclusion

$$((\pi_1\ (1,\ 2)),\ (\pi_2\ (1,\ 2)))$$

# Tuples



$( (\pi_1\ (1,\ 2)),\ (\pi_2\ (1,\ 2)))$

$( 1,\ (\pi_2\ (1,\ 2)))$

# Tuples



$$(\,1\,,\ (\pi_2\ (1\,,\ 2)))$$

# Tuples





$(\boxed{1}, (\pi_2 (1, 2)))$

$(1, (\pi_2 (1, 2)))$

# Tuples



$(1\,,\ (\pi_2\ (1,\ 2)))$

# Tuples



$$(1, \boxed{(\pi_2\ (1,\ 2))})$$

$$(1, \boxed{2})$$

# Tuples



```
(1, 2)
```

# Tuples

```
(1, 2)
```

```
(1, 2)
```

# Discriminated unions

# Discriminated unions



- A choice of values is defined simply as something that stores either of two possible values
- We call such a choice a **discriminated union**
- We build a discriminated union with either of two functions to build the first or the second value
- They are usually called `inl` and `inr`[a]

[a] *in* stands for injection, and *l* and *r* stand for left and right

$(\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))$

$(\lambda y \rightarrow (\lambda f\ g \rightarrow (g\ y)))$

# Discriminated unions



```
(inl 1)
```

# Discriminated unions



```
(inl 1)
```

```
(inl 1)
```

# Discriminated unions

```
( inl  1)
```

# Discriminated unions



```
( inl  1)
```

```
( (λx→ (λf g→(f x)))  1)
```

# Discriminated unions



```
((λx→ (λf g→(f x))) 1)
```

# Discriminated unions



```
((λx→ (λf g→(f x))) 1)
```

```
((λx→ (λf g→(f x))) 1)
```

# Discriminated unions



```
((λx→ (λf g→(f x))) 1)
```

# Discriminated unions



```
((λx→ (λf g→(f x))) 1)
```

```
(λf g→(f 1))
```

# Discriminated unions



```
(λf g→(f 1))
```

# Discriminated unions



```
(λf g→(f 1))
```

```
(λf g→(f 1))
```

# Discriminated unions



$(\lambda$f g$\rightarrow$(f 1))

# Discriminated unions



```
(λf g→(f 1))
```

```
(inl 1)
```

# Discriminated unions

Data structures

The INFDEV@HR Team

Introduction

Let-in

Data types

Tuples

Discriminated unions

Conclusion

```
(inr TRUE)
```

# Discriminated unions



```
(inr TRUE)
```

```
(inr TRUE)
```

# Discriminated unions

```
( inr  TRUE)
```

# Discriminated unions



```
( inr  TRUE)
```

```
( (λy→ (λf g→(g y)))  TRUE)
```

# Discriminated unions



```
((λy→ (λf g→(g y))) TRUE)
```

# Discriminated unions



```
((λy→ (λf g→(g y))) TRUE)
```

```
((λy→ (λf g→(g y))) TRUE)
```

# Discriminated unions



$$((\lambda y \rightarrow (\lambda f\ g \rightarrow (g\ y)))\ \mathtt{TRUE})$$

# Discriminated unions



```
((λy→ (λf g→(g y))) TRUE)
```

```
(λf g→(g TRUE))
```

# Discriminated unions



```
(λf g→(g TRUE))
```

$(\lambda$f g$\rightarrow$(g TRUE))

$(\lambda$f g$\rightarrow$(g TRUE))

# Discriminated unions



```
(λf g→(g TRUE))
```

# Discriminated unions



```
(λf g→(g TRUE))
```

```
(inr TRUE)
```

# Discriminated unions



- Extracting the input of a discriminated union is a process known as `match`[a]
- Given a union and two functions (one per case), if the union was the first case we apply the first function, otherwise we apply the second function

[a] which is a sort of `switch`, just on steroids

$$(\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))$$

$$\texttt{(((match (inl 1)) (}\lambda\texttt{x}\rightarrow\texttt{(x + 1))) (}\lambda\texttt{y}\rightarrow\texttt{(y }\wedge\texttt{ FALSE)))}$$

# Discriminated unions



```
(((match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

```
(((match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

# Discriminated unions

$$\texttt{(((match (inl 1)) (}\lambda\texttt{x}\rightarrow\texttt{(x + 1))) (}\lambda\texttt{y}\rightarrow\texttt{(y}\ \wedge\ \texttt{FALSE)))}$$

# Discriminated unions



```
((( match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

```
((( (λu→ (λf g→((u f) g))) (inl 1)) (λx→(x + 1)
    )) (λy→(y ∧ FALSE)))
```

# Discriminated unions

$$\boxed{\begin{array}{l}\texttt{((((}\lambda\texttt{u}\rightarrow\ \texttt{(}\lambda\texttt{f g}\rightarrow\texttt{((u f) g))) (inl 1)) (}\lambda\texttt{x}\rightarrow\texttt{(x} + \\ \quad \texttt{1))) (}\lambda\texttt{y}\rightarrow\texttt{(y} \wedge \texttt{FALSE)))}\end{array}}$$

# Discriminated unions



$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\texttt{inl}\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\boxed{\texttt{inl}}\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

# Discriminated unions

```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
      1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) ( inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (
    (λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

# Discriminated unions



$$\boxed{((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ ((\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))}$$

# Discriminated unions



$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ ((\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ ((\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

# Discriminated unions

```
((((λu→ (λf g→((u f) g)))
    ((λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→
    (y ∧ FALSE)))
```

# Discriminated unions



```
((((λu→ (λf g→((u f) g)))
    ((λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→
    (y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx
    →(x + 1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions

Data structures

The INFDEV@HR Team

Introduction

Let-in

Data types

Tuples

Discriminated unions

Conclusion

$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\lambda f\ g \rightarrow (f\ 1)))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions

$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\lambda f\ g \rightarrow (f\ 1)))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions



$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\mathtt{inl}\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \land \mathtt{FALSE})))$$

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
     1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
      1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions

```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (
    (λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

# Discriminated unions



$$\boxed{\texttt{((((}\lambda\texttt{u}\rightarrow\texttt{ (}\lambda\texttt{f g}\rightarrow\texttt{((u f) g))) ((}\lambda\texttt{x}\rightarrow\texttt{ (}\lambda\texttt{f g}\rightarrow\texttt{(f x))) 1)) (}\lambda\texttt{x}\rightarrow\texttt{(x + 1))) (}\lambda\texttt{y}\rightarrow\texttt{(y }\wedge\texttt{ FALSE)))}}$$

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) ((λx→ (λf g→(f x))
   ) 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g)))
   ((λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→
   (y ∧ FALSE)))
```

# Discriminated unions



$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ ((\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

# Discriminated unions



$$((((\lambda u\rightarrow\ (\lambda f\ \ g\rightarrow((u\ \ f)\ \ g)))\ \ ((\lambda x\rightarrow\ (\lambda f\ \ g\rightarrow(f\ \ x)))\ \ 1)\ )\ \ (\lambda x\rightarrow(x\ +\ 1)))\ \ (\lambda y\rightarrow(y\ \wedge\ FALSE)))$$

$$((((\lambda u\rightarrow\ (\lambda f\ \ g\rightarrow((u\ \ f)\ \ g)))\ \ (\lambda f\ \ g\rightarrow(f\ \ 1)))\ \ (\lambda x\rightarrow(x\ +\ 1)))\ \ (\lambda y\rightarrow(y\ \wedge\ FALSE)))$$

# Discriminated unions

$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\lambda f\ g \rightarrow (f\ 1)))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions

```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions



$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\texttt{inl}\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

# Discriminated unions



```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions

$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\boxed{\texttt{inl}}\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

# Discriminated unions



$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ (\texttt{inl}\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

$$((((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ ((\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

# Discriminated unions



$$\texttt{((((}\lambda\texttt{u}\rightarrow\ \texttt{(}\lambda\texttt{f g}\rightarrow\texttt{((u f) g))) ((}\lambda\texttt{x}\rightarrow\ \texttt{(}\lambda\texttt{f g}\rightarrow\texttt{(f x))) 1)) (}\lambda\texttt{x}\rightarrow\texttt{(x}+\texttt{1))) (}\lambda\texttt{y}\rightarrow\texttt{(y}\wedge\texttt{ FALSE)))}$$

# Discriminated unions

Data structures

The INFDEV@HR Team

Introduction

Let-in

Data types

Tuples

Discriminated unions

Conclusion

```
((((λu→ (λf g→((u f) g))) ((λx→ (λf g→(f x))
    ) 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

```
((
    ((λu→ (λf g→((u f) g))) ((λx→ (λf g→(f x))) 1))
    (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions



$$
\begin{array}{l}
((\\
\quad ((\lambda u \rightarrow\ (\lambda f\ g \rightarrow ((u\ f)\ g)))\ ((\lambda x \rightarrow\ (\lambda f\ g \rightarrow (f\ x)))\ 1))\\
\quad (\lambda x \rightarrow (x\ +\ 1)))\ (\lambda y \rightarrow (y\ \wedge\ \mathtt{FALSE})))
\end{array}
$$

# Discriminated unions



$$((\ ((\lambda u \rightarrow (\lambda f\ g \rightarrow ((u\ f)\ g)))\ ((\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

$$(((\lambda f\ g \rightarrow ((\ ((\lambda x \rightarrow (\lambda f\ g \rightarrow (f\ x)))\ 1)\ f)\ g))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

# Discriminated unions



$$\texttt{(((}\lambda\texttt{f g}\rightarrow\texttt{((((}\lambda\texttt{x}\rightarrow\texttt{ (}\lambda\texttt{f g}\rightarrow\texttt{(f x))) 1) f) g)) (}\lambda\texttt{x}\rightarrow\texttt{ (x + 1))) (}\lambda\texttt{y}\rightarrow\texttt{(y }\wedge\texttt{ FALSE)))}$$

# Discriminated unions



$$(((\lambda f\ \ g\rightarrow((((\lambda x\rightarrow\ (\lambda f\ \ g\rightarrow(f\ \ x)))\ \ 1)\ \ f)\ \ g))\ \ (\lambda x\rightarrow (x\ +\ 1)))\ \ (\lambda y\rightarrow(y\ \wedge\ \mathtt{FALSE})))$$

$$( \\ ((\lambda f\ \ g\rightarrow((((\lambda x\rightarrow\ (\lambda f\ \ g\rightarrow(f\ \ x)))\ \ 1)\ \ f)\ \ g))\ \ (\lambda x\rightarrow(x\ +\ 1 \\ (\lambda y\rightarrow(y\ \wedge\ \mathtt{FALSE})))$$

# Discriminated unions

```
(
    ((λf g→((((λx→ (λf g→(f x))) 1) f) g)) (λx→(x + 1
    (λy→(y ∧ FALSE)))
```

# Discriminated unions



```
(
    ((λf g→((((λx→ (λf g→(f x))) 1) f) g)) (λx→(x + 1
    (λy→(y ∧ FALSE)))
```

```
((λg→((((λx f g→(f x)) 1) (λx→(x + 1))) g))
    (λy→(y ∧ FALSE)))
```

# Discriminated unions



```
((λg→((((λx f g→(f x)) 1) (λx→(x + 1))) g))
    (λy→(y ∧ FALSE)))
```

# Discriminated unions



```
((λg→((((λx f g→(f x)) 1) (λx→(x + 1))) g))
    (λy→(y ∧ FALSE)))
```

```
((λg→((((λx f g→(f x)) 1) (λx→(x + 1))) g)) (λy→(y ∧
```

# Discriminated unions

$$((\lambda g \rightarrow ((((\lambda x\ f\ g \rightarrow (f\ x))\ 1)\ (\lambda x \rightarrow (x + 1)))\ g))\ (\lambda y \rightarrow (y \wedge$$

$((\lambda g \rightarrow ((((\lambda x\ f\ g \rightarrow (f\ x))\ 1)\ (\lambda x \rightarrow (x + 1)))\ g))\ (\lambda y \rightarrow (y \wedge$

$((((\lambda x\ f\ g \rightarrow (f\ x))\ 1)\ (\lambda x \rightarrow (x + 1)))$
$\quad (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$

# Discriminated unions

```
((((λx f g→(f x)) 1) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

# Discriminated unions



```
((((λx f g→(f x)) 1) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

```
((((λx f g→(f x)) 1) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

```
(( ((λx f g→(f x)) 1) (λx→(x + 1))) (λy→(y ∧
   FALSE)))
```

# Discriminated unions

```
(( ((λx f g→(f x)) 1) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

```
(((λf g→(f 1 )) (λx→(x + 1))) (λy→(y ∧ FALSE
    )))
```

# Discriminated unions



$$(((\lambda f\ g \rightarrow (f\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$$

# Discriminated unions



```
(((λf g→(f 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)
    ))
```

```
( ((λf g→(f 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

# Discriminated unions



$$( ((\lambda f\ g \rightarrow (f\ 1))\ (\lambda x \rightarrow (x + 1)))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

# Discriminated unions



$$\texttt{(}\ \texttt{((}\lambda\texttt{f g}\rightarrow\texttt{(f 1)) (}\lambda\texttt{x}\rightarrow\texttt{(x + 1)))}\ \texttt{(}\lambda\texttt{y}\rightarrow\texttt{(y}\ \wedge\ \texttt{FALSE)))}$$

$$\texttt{((}\lambda\texttt{g}\rightarrow\texttt{( (}\lambda\texttt{x}\rightarrow\texttt{(x + 1)) 1)) (}\lambda\texttt{y}\rightarrow\texttt{(y}\ \wedge\ \texttt{FALSE)))}$$

# Discriminated unions



$$((\lambda g \rightarrow ((\lambda x \rightarrow (x + 1))\ 1))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

# Discriminated unions



```
((λg→((λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

```
((λg→((λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

$$((\lambda g \rightarrow ((\lambda x \rightarrow (x + 1))\ 1))\ (\lambda y \rightarrow (y \wedge \texttt{FALSE})))$$

# Discriminated unions

$((\lambda g \rightarrow ((\lambda x \rightarrow (x + 1))\ 1))\ (\lambda y \rightarrow (y \wedge \mathtt{FALSE})))$

$((\lambda x \rightarrow (x + 1))\ 1)$

# Discriminated unions



$$((\lambda x \rightarrow (x + 1))\ 1)$$

# Discriminated unions



$((\lambda x \rightarrow (x + 1))\ 1)$

$((\lambda x \rightarrow (x + 1))\ 1)$

# Discriminated unions



```
((λx→(x + 1)) 1)
```

# Discriminated unions



```
((λx→(x + 1)) 1)
```

```
(1 + 1)
```

Data structures

The INFDEV@HR Team



# Discriminated unions

```
(1 + 1)
```

# Discriminated unions



```
(1 + 1)
```

```
(1 + 1)
```

# Discriminated unions



```
(1 + 1)
```

# Discriminated unions



```
(1 + 1)
```

```
2
```

# Discriminated unions



## Choice between a pair of values

We should expect that `inl` and `inr` are inverse operations to `match`

# Discriminated unions

```
(((match (inl 1)) inl) inr)
```

# Discriminated unions

```
(((match (inl 1)) inl) inr)
```

```
(((match (inl 1)) inl) inr)
```

# Discriminated unions

```
(( (match (inl 1)) inl) inr)
```

# Discriminated unions



```
(( (match (inl 1)) inl) inr)
```

```
(inl 1 )
```

# Discriminated unions



```
(((match (inr TRUE)) inl) inr)
```

# Discriminated unions



```
(((match (inr TRUE)) inl) inr)
```

```
(( (match (inr TRUE)) inl) inr)
```

# Discriminated unions

```
(( (match (inr TRUE)) inl) inr)
```

# Discriminated unions



```
(( (match (inr TRUE)) inl) inr)
```

```
(inr TRUE )
```

# Conclusion

# Conclusion



## Recap

- Lambda terms can be used to encode arbitrary basic data types
- The terms are always lambda expression which, when they get parameters passed in, identify themselves somehow
- Identification can be done by applying something (possibly even a given number of times), or returning one of the parameters

# Conclusion



## Recap

- The data types we have seen cover an impressive range of applications
- Tuples cover grouping data together (like the fields of a class)
- Unions cover choosing different things (like the polymorphism of an interface that might be implemented by various concrete classes)
- Combining these two covers all possible programming needs, even for more complex data structures

# This is it!

The best of luck, and thanks for the attention!