

# Functional Programming

## Type definitions

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2017-2018

# Type aliases

Explaining the meaning of data in comments is bad!

Introduce new, self explaining types.

```
type Name = String
```

```
type Title = String
```

```
type Year = Int
```

```
type Age = Int
```

```
type User = (Name,      Year)
```

```
--           ^ name      ^ Year of birth
```

```
type Film = (Title, Age)
```

```
--           ^ fsk
```

```
type Purchase = (Name, Title, Year) -- <---+
```

```
--           ^ user name ^ item name      ^ date of purchase
```

```
users :: [User]
```

# Datatypes

## Example scenario

- model a card game (hearts)
- represent the game items!
- define game logic on the representations!

## Intermezzo: The game

### Microsoft Hearts (Wikipedia link)

- computer game based on card game “Hearts”
- included in Windows 3.1 through Windows 7
- discontinued

# The game

## Gameplay

- Four players (three simulated)
- Trick-taking game
- Each player plays one card to a trick
- Trick won by highest card of the suit led; no trump
- Suit must be followed
- Heart cannot lead until
  - ▶ either Heart has been broken — a player played Heart
  - ▶ or the leading player has only Heart
- Points are scored by any Hearts (1 point) and the Queen of Spades (13 points)

# The game

## Gameplay

- Four players (three simulated)
- Trick-taking game
- Each player plays one card to a trick
- Trick won by highest card of the suit led; no trump
- Suit must be followed
- Heart cannot lead until
  - ▶ either Heart has been broken — a player played Heart
  - ▶ or the leading player has only Heart
- Points are scored by any Hearts (1 point) and the Queen of Spades (13 points)

## Objective

- Avoid gaining points **or** gain all 26 points

# Data model for card games

- A card has a **Suit** and a **Rank**
- A card beats another card if it has the same suit, but higher rank
- Todo:
  - ▶ represent cards
  - ▶ define when one card beats another
  - ▶ define a function that chooses a beating card from a hand of cards, if possible

# Define new data types

A card has a Suit

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

## Explanation

- new type consisting of (exactly) four values
- Suit: the name of the new type
- Spades, Hearts, ...: the names of its **constructors**.
- Type and constructor names must be capitalized



# Printing new data types

```
Main> Spades
```

```
<interactive>:3:1:
```

```
No instance for (Show Suit) arising from  
a use of 'print'
```

```
Possible fix: [...]
```

Oops!

- Haskell does not know how to print a Suit
- but we can ask for a default (or write our own printer)

## Printing derived

```
data Suit = Spades | Hearts | Diamonds | Clubs
  deriving (Show) -- makes 'Suit' printable
```

Defines a function `show` for `Suit`, which is automatically called by Haskell's printer

```
Main> Spades
```

```
Spades
```

```
Main> show Spades
```

```
"Spades"
```

```
Main> :t show
```

```
show :: Show a => a -> String
```

# Functions on data types

Each suit has a color:

```
data Color = Black | Red  
  deriving (Show)
```

Define a color function by pattern matching

```
color :: Suit -> Color  
color = undefined
```

## More data

A card has a suit and a **rank**:

```
data Rank = Numeric Integer | Jack | Queen
          | King | Ace
  deriving Show
```

The constructor `Numeric` is different: it takes an argument.

```
Main> :t Numeric
Numeric :: Integer -> Rank
```

# Comparing ranks

```
-- |rankBeats r1 r2  
-- returns True, if r1 beats r2  
rankBeats :: Rank -> Rank -> Bool  
rankBeats r1 r2 = undefined
```

To continue, we need an ordering on ranks

## Ordering ranks by pattern matching

```
-- rankBeats r1 r2 returns True, if r1 beats r2
rankBeats :: Rank -> Rank -> Bool

rankBeats _ Ace      = False
rankBeats Ace _      = True
rankBeats _ King     = False
rankBeats King _     = True
rankBeats _ Queen    = False
rankBeats Queen _    = True
rankBeats _ Jack     = False
rankBeats Jack _     = True
rankBeats (Numeric n1) (Numeric n2) = n1 > n2
--      ^^ pattern match on constructor
--      yields its argument
```

# Cards, finally

A card has a Suit and a Rank

```
data Card = Card Rank Suit
  deriving (Show)
```

```
rank :: Card -> Rank
rank (Card r s) = r
```

```
suit :: Card -> Suit
suit (Card r s) = s
```

- single constructor with two parameters
- (in principle, a tuple with a special name)
- rank, suit are **selector functions**

# Comparing Cards

A card **beats** another card, if it has the same suit, but a higher rank

```
cardBeats :: Card -> Card -> Bool
cardBeats givenCard c = suit givenCard == suit c
                        && rankBeats (rank givenCard)
                                   (rank c)
```



# Hand of Cards

```
type Hand = [Card]
```

```
chooseCard :: Card -> Hand -> Card
```

```
chooseCard givenCard h = undefined
```

To develop chooseCard refine h by pattern matching

## Choose a card

```
type Hand = [Card]
```

```
chooseCard :: Card -> Hand -> Card
```

```
chooseCard givenCard [] = undefined -- ???
```

```
chooseCard givenCard (x:xs) = undefined
```

- What should we do if the hand is empty?
- Avoid by defining only non-empty hands!

# Non-empty hands

```
data Hand = Last Card | Next Card Hand
  deriving (Show, Eq)
```

- Recursive datatype definition
- Last Card is the **base case**

## Get card from non-empty hand

- A Hand is never empty
- Thus we can always obtain a card

```
topCard :: Hand -> Card
```

```
topCard (Last c) = c
```

```
topCard (Next c _) = c
```

## Choosing from non-empty hand

```
-- choose a beating card, if possible  
chooseCard :: Card -> Hand -> Card  
chooseCard = undefined
```

## Choosing from non-empty hand

```
-- choose a beating card, if possible
chooseCard :: Card -> Hand -> Card
chooseCard gc (Last c) = c -- may beat, or not
chooseCard gc (Next c h) | cardBeats gc c = c
                        | otherwise       = chooseCard gc h
```

## Break Time — Questions?

