GÖTEBORGS UNIVERSITET
CHALMERS

Introduction to Functional Programming

Modelling with data types, and list functions

Some slides are based on Graham Hutton's public slides

Recap previous lecture

- Guarded equations (cases)
- Recursion, again

- Types
- Lists and tuples

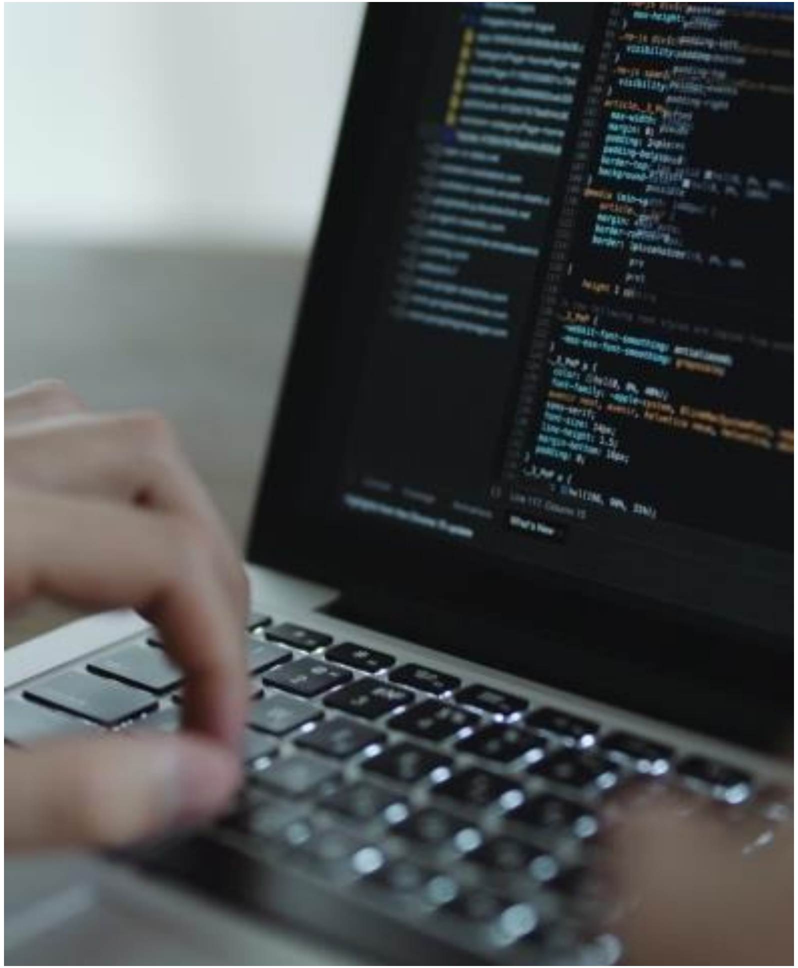
Today

- Another student representative GU

- List comprehensions
- Modelling with data types
- The ‘cons’ operator
- Defining (recursive) functions over lists
- `where`-clauses and `let`-expressions
- (Testing with properties)

MODELLING DATA

GÖTEBORGS UNIVERSITET
CHALMERS



Modelling data

- A big part of designing software
  is modelling the data in an appropriate way.
- A lot of data can be represented as numbers or lists or other predefined types, but…
- Today's lecture: we look at how to model data by defining new types in Haskell.
  - This makes the much code clearer and helps prevent certain mistakes that otherwise easily happen.

Type declarations

- In Haskell, a new name for an existing type can be defined using a *type declaration*.

- Type declarations can be used to make other types easier to read.

- Type declarations can be nested
  - They cannot be recursive

`String` is a *synonym* for the type `[Char]`

```
type String = [Char]



type Pos = (Int, Int)

origin :: Pos
origin = (0, 0)

left :: Post -> Pos
left (x, y) = (x - 1, y)


type Trans = Pos -> Pos
type Tree = (Int, [Tree])
```

Data declarations

- A completely new type can be defined by specifying its values using a *data declaration*.

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

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

- The two values `False` and `True` are called the constructors for the type `Bool`.
- Type and constructor names must always begin with an upper-case letter.
- Data declarations are like context free grammars. The former specifies the values of a type, the latter the sentences of a language.

Using data declarations

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

```
-- Given:
data Answer = Yes | No | Unknown


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


-- We can define
flip :: Answer -> Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

GÖTEBORGS UNIVERSITET

CHALMERS

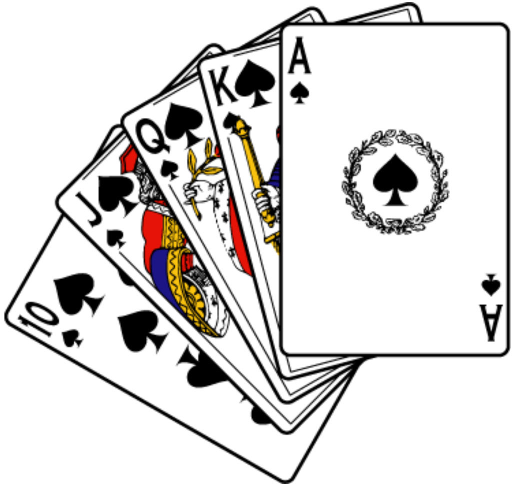
Storing data

- The constructors in a data declaration can also have parameters.
- These are also called (constructor) fields
  - We create these fields by listing the type of the field after the constructor name
- We use these files to 'store' data in a constructor
  - For example, we store a single floating-point number (of type `Float`) in the `Circle` constructor
- `Circle` and `Rect` can be viewed as functions that construct values of type `Shape`

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

square :: Float -> Shape
square n = Rect n n

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


-- Circle has type Float -> Shape
-- Rect has type Float -> Float -> Shape
```

Modelling a card game

- Consider playing cards used in card games
- Every card has a suit: ♠ ♥ ♦ ♣
- We can define a new data type for suits:



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

The new type

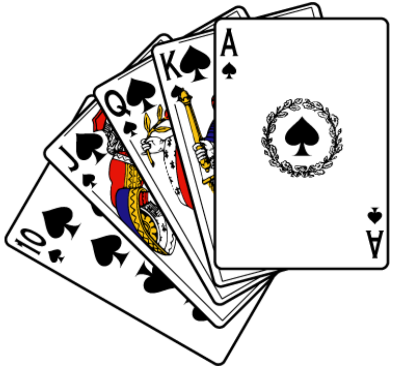
The values of this type

Types and constructors

The new type

The values of this type

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



- Interpretation:
  - *"Here is a new type* `Suit`*. This type has four possible values:* `Spades`, `Hearts`, `Diamonds` *and* `Clubs`*."*

- This definition introduces five things:
  - The type `Suit` and
  - four constructors (`Spades :: Suit`, `Hearts :: Suit`, ...)

Types and constructors

Type

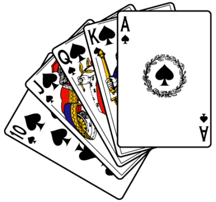
Type

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

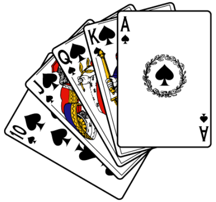
Constructor

Constructor

- This definition introduces six things:
  - The type `Rank`
  - The constructors:
    - `Ace     :: Rank`
    - `King    :: Rank`
    - `Queen   :: Rank`
    - `Jack    :: Rank`
    - `Numeric :: Integer -> Rank`



Types and constructors



Type

```
data Card = Card Rank Suit
```

Type

Type

Constructor

- This definition introduces two things:
  - The type `Card`
  - The constructor
    - `Card :: Rank -> Suit -> Card`

```
data Card = Card
  { rank :: Rank
  , suit :: Suit
  }
```

Alternative (record) syntax

GÖTEBORGS UNIVERSITET

CHALMERS

Pattern matching

- Functions on the values of a data type are usually defined by *pattern matching*
- Functions will often have one equation for each alternative in the data type
- Sometimes alternatives can be combined by using variable or wildcard (don't-care) patterns

```
colour :: Suit -> Colour
colour Spades = Black
colour Clubs  = Black
colour _      = Red


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

FUNCTIONS ON LISTS

Prelude functions on lists

- Haskell comes with a large number of standard library functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on *lists*.

```
ghci> head [1,2,3,4,5]
1
```

Select the first element of a list

```
ghci> tail [1,2,3,4,5]
[2,3,4,5]
```

Remove first element from a list

```
ghci> [1,2,3,4,5] !! 2
3
```

Select the n-th element of a list

```
ghci> take 3 [1,2,3,4,5]
[1,2,3]
```

Select first n elements of a list

```
> drop 3 [1,2,3,4,5]
[4,5]
```

Remove first n elements from a list

Prelude functions on lists

- Haskell comes with a large number of standard library functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on *lists*.

```
ghci> length [1,2,3,4,5]
5

ghci> sum [1,2,3,4,5]
15

ghci> product [1,2,3,4,5]
120

ghci> [1,2,3] ++ [4,5]
[1,2,3,4,5]

> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Calculate the length of a list

Calculate the sum of all elements

Calculate the product of all elems

Append two lists

Reverse a list

List patterns

- Internally, every non-empty list is constructed by repeated use of an operator `(:)` called “cons” that adds an element to the start of a list.

```
[1,2,3,4]

= means =>

1 : (2 : (3 : (4 : [])))
```

- Functions on lists can be defined using `x:xs` patterns
  - `head` and `tail` map any *non-empty* list to its first and remaining elements.

```
head (x:_) = x

tail (_:xs) = xs
```

List patterns

- `x:xs` patterns only match *non-empty lists*

```
ghci> head []
*** Exception: empty list
```

- `x:xs` patterns must be *parenthesised*, because application has priority over `(:)`.
  - For example, the definition of `head` on the right gives an error

```
head x:_ = x
```

Recursion on lists

- Recursion is not restricted to numbers, but can also be used to define functions on *lists*.
- The `product` function maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

```
product []     = 1
product (n:ns) = n * product ns
```

```
  product [2,3,4]
= 2 * product [3,4]
= 2 * (3 * product [4])
= 2 * (3 * (4 * product []))
= 2 * (3 * (4 * 1))
= 24
```

Recursion on lists

- Using the same pattern of recursion as in product we can define the `length` function on lists.

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

```
length []     = 0
length (_:xs) = 1 + length xs
```

```
  length [1,2,3]
= 1 + length [2,3]
= 1 + (1 + length [3])
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 3
```

Recursion on lists

- Using a similar pattern of recursion we can define the `reverse` function on lists.

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]


  reverse [1,2,3]
= reverse [2,3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= ((reverse [] ++ [3]) ++ [2]) ++ [1]
= (([] ++ [3]) ++ [2]) ++ [1]
= [3,2,1]
```

Local definitions

- In Haskell we have two ways of making local definitions:
  - `where`-clauses
  - `let`-expressions
- This useful for helper functions and being able to reuse the same name
  - Finding good names is hard
- Lexical scope: the definition that is 'closest' is chosen.
  - For example: `inc` from defined in the `where`/`let` has precedence over the top-level `inc`

```
inc x = x + 10

addN :: Int -> [Int] -> [Int]
addN n xs = [inc x | x <- xs]
 where
  inc x = x + n

addM :: Int -> [Int] -> [Int]
addM m xs =
  let inc x = x + m in  [inc x | x <- xs]
```

GÖTEBORGS
UNIVERSITET

CHALMERS