# Advanced Programming 2022

## Introduction to Haskell

Andrzej Filinski
andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 6, 2022

# Let's begin!

The purpose of this course is to provide practical experience with sophisticated programming techniques and paradigms from a language-based perspective. The focus is on high-level programming and systematic construction of well-behaved programs.

– `https://kurser.ku.dk/course/ndaa09013u/2022-2023`

Why would you learn a new programming language

(let alone several)?

# A Language-Based Perspective

Different languages offer:

- Different levels of abstraction
  - Contrast assembly, C, and Python
- Different assurances
  - Static (compile-time) analyses
  - Dynamic (run-time) checking
- Different programming models
  - Functional vs. imperative vs. declarative programming
  - Lazy evaluation vs. eager evaluation vs. proof search
  - Message passing vs. shared memory (read-only or read/write)
- Different primitives, libraries, and frameworks

Exposure to variety makes you a more effective programmer, *in any language*.

# Why Haskell?

- Modern functional programming (FP) language
  - Introduced $\sim$1990, but has been evolving continuously since
  - Vibrant user and developer community
  - Good cross-platform support
  - Directly used in growing number of application domains
- Useful medium to present general programming abstractions and principles
  - Easier to explain many ideas in a functional setting
  - Many FP concepts and techniques steadily diffusing into "mainstream" languages
- Goal of course is *not* to make you Haskell experts
  - "Program *into* a language, not *in* it." –D.Gries ($\sim$1981)
  - Do exploit useful constructs and idioms of host language, but don't let it constrain your high-level thinking.

# Haskell fundamentals

- *Value-oriented* (*applicative*) programming paradigm
  - Will see others later in the course
- Main computation model: *evaluation* of *expressions*
  - Not sequential *execution* of *statements*
    - Though that can be accommodated as a special case
  - *Purely* functional
    - No hidden/silent side effects at all
- Strongly, statically typed
  - Surprisingly many problems caught at compile time
- If you already know another typed functional language (SML, OCaml, F#), today will be mainly a refresher
  - Next time: Haskell-specific concepts and constructs
- If not, don't panic!
  - Basic concepts are really quite simple

# Types

- Haskell (like Java, unlike C) is *strongly* typed.
  - Types enforce *abstractions*, both language-provided and programmer-defined.
  - Cannot *construct* “ill-formed” values of a type.
    - No crashes/segfaults (“casting `int` to pointer”)
    - No violation of data-structure invariants
  - Cannot even *observe* interior structure of data values, except through designated API.
    - No inspecting of heap/stack layout (“casting pointer to `int`”)
    - No hidden dependencies on particular implementation
- Haskell (like C, unlike Python) is *statically* typed.
  - Only well-typed programs may even be run.
  - Type system is very flexible, normally unobtrusive.
    - A reported type error *almost always* reflects logical error in program, not weakness/deficiency of type checker
    - Once program type-checks, usually close to working

# Types and values

- Types classify values.
  - Notation: *value* `::` *type*
- Usual complement of *basic* types, including:
  - Integers: `3 :: Int`, `43252003274489856000 :: Integer`
  - Floating point: `2.718281828 :: Double` (`Float` rarely used)
  - Booleans: `True :: Bool`
  - Characters: `'x' :: Char`
  - Strings: `"new\nline" :: String`
    - Actually, `type String = [Char]` (list of characters)
- *Compound* types, including:
  - Tuples: `(2, 3.4, False) :: (Int, Double, Bool)`
  - Lists (homogeneous): `[2,3,5,7] :: [Int]`
  - May be nested:
    `([(1, 2.0), (3, 4.0)], True) :: ([(Int, Double)], Bool)`

# Expression evaluation

- *Expressions* also have types
  - The expression `2+2 :: Int` evaluates to the value `4 :: Int`
- *Type safety*: expression of a given type always evaluates to value of that type.
  - Or possibly a runtime error (e.g., div. by 0), or nontermination
  - Far from trivial to show, given advanced features in Haskell's type system.
- Haskell implementations generally provide an interactive mode
  - Traditionally called a read-eval-print loop (REPL)
  - In Glasgow Haskell Compiler (GHC), invoked as `ghci -W`
    - The `-W` enables useful warnings; omit at your peril!
  - When using Stack, try `alias ghci='stack exec ghci -- -W'` (or equivalent in your favorite shell).

# Using the REPL environment

- Evaluating expressions:

```
ghci> "foo" ++ "bar"
"foobar"
ghci> head "foo"
'f'
ghci> head ""
*** Exception: Prelude.head: empty list
```

- Can also type-check expressions without evaluating:

```
ghci> :type head ""
head "" :: Char
```

- Useful for debugging and experimentation, but not meant for writing actual programs.
  - Load a set of definitions from a file, then experiment interactively.

# Expression forms

- Expressions are built up from
  - *Literals* (atomic values): `42`, `'x'`, `True`
  - Constructors of *structured* values: `[3,4]`, `(5,'6',"7")`
  - Constants and variables (global or local):
    `pi`, `let x = 3 in x*x`
  - Function calls (prefix and infix): `sqrt 4.0`, `5 + 6`
  - Conditionals: `if x > y then x else y`
    - Later generalized to `case`-expressions
- Large number of built-in constants and functions.
  - Most common ones are always available (standard prelude)
  - Others must be `import`ed from relevant module first
  - Hoogle (`https://hoogle.haskell.org/`) is your friend!
- Can add own definitions:
  - At top level (usually only one-liners)

    ```
    ghci> courseName = "Advanced Programming"
    ghci> wordCount s = length (words s)
    ```
  - In separate file (next slide)

# Definitions in separate file

- Can naturally stretch over multiple lines
- Should always include explicit type signatures for all definitions
  - Not formally required (Haskell can almost always infer them), but makes it *much* easier to read and understand your code.
- Example: in file `mydefs.hs`

```haskell
courseName :: String
courseName = "Advanced Programming"

wordCount :: String -> Int
wordCount s = length (words s)
```

- Can load from top-level loop

```
ghci> :load mydefs.hs
[...]
ghci> wordCount courseName
2
```

- Later: code in files should be organized into *modules*.

# More about Haskell definitions

- Haskell syntax (like Python, F#) is indentation-sensitive
  - Always use spaces, not tabs!
    - Configure your editor to insert spaces on tab-key presses
- Multiple definitions in a group must start at same level:

```
let a = ...
    f x = ...
in ...
```

  - Or separate with ";": `let a = ... ; f x = ... in ...`
- *Increase* indentation to continue previous line

```
double x =
  x + x
```

- All definitions (whether local or global) may be *mutually recursive*:

```
isEven, isOdd :: Int -> Bool
isEven x = if x == 0 then True else isOdd (x - 1)
isOdd x = if x == 0 then False else isEven (x - 1)
```

# More about Haskell functions

- Functions are values, too, but cannot be printed.

```
ghci> :t wordCount
wordCount :: String -> Int
ghci> wordCount
<interactive>:6:1: No instance for (Show (String -> Int)) ...
```

- Functions may have multiple arguments:

```
addt :: (Int, Int) -> Int  -- tupled style
addt (x, y) = x + y

addc :: Int -> Int -> Int  -- curried style (preferred)
addc x y = x + y     -- [named for Haskell Brooks Curry]
```

- Functions may also take *other functions* as arguments:

```
ghci> map isOdd [2,3,5]
[False,True,True]
```

# Anonymous functions

- Can construct functional values without naming them:

```
ghci> map (\x -> x+3) [2,3,5]
[5,6,8]
```

- "\" pronounced "lambda" (here): ASCII approximation of "$\lambda$".
  - In fact, in typeset/pretty-printed Haskell code, you may see the above rendered as "$\mathit{map}\ (\lambda x \rightarrow x + 3)\ [2, 3, 5]$".
- Could define previous functions more explicitly as:

```
addt :: (Int, Int) -> Int   -- tupled style
addt = \p -> fst p + snd p

addc :: Int -> (Int -> Int) -- curried style
addc = \x -> \y -> x + y
```

- Note: `addc 3` actually returns the function `\y -> 3 + y`.
  - `addc 3 4` $\simeq$ `(\y -> 3 + y) 4` $\simeq$ `3 + 4` $\simeq$ `7`.

# Infix operators

- Haskell makes no fundamental distinction between *functions* and *operators*, after lexing/parsing.
- Two syntactic classes of identifiers:
  - Alphanumeric (prefix): any seq. of letters, digits, underscores, primes (')
    - ... except a few *reserved* words, e.g., `let`
    - Must *start* with lowercase letter or underscore
    - Conventional style: `longName`, not `long_name`
  - Symbolic (infix): any seq. of special characters (`!`, `#`, `$`, `+`, `:`, ...)
    - ... except a few reserved operators, e.g., `->`
    - Must *not start* with a colon
- Can use any operator as (two-argument) function by enclosing in parentheses: `(+) 2 3` evaluates to `5`.
- Conversely, can use any two-argument function as operator by enclosing in backticks: ``10 `mod` 4`` evaluates to `2`.
  - Can specify desired precedence and/or associativity for non-standard operators with `infix`{`l`,`r`,} declarations.

# Polymorphism

- Functions (and other values) may be *polymorphic*:
  - Have type *schemas*, where some concrete types have been replaced by (lowercase) *type variables*

    ```
    dup :: a -> (a, a)
    dup x = (x, x)
    ```
  - Type system will automatically *instantiate* such types to match use context:
    - `dup 5` evaluates to `(5, 5)` (taking `a` = `Int`)
    - `dup True` evaluates to `(True, True)` (taking `a` = `Bool`).
    - ...
- Sometimes polymorphism limited to certain *classes* of types:
  - Numeric types: `Int`, `Double`, ...
    - `(+) :: Num a => a -> a -> a`
    - `2 + 3` evaluates to `5`
    - `2.0 + 3.0` evaluates to `5.0`
    - `"2" + "3"` is a type error
  - Equality types: almost all except functions
    - `(==) :: Eq a => a -> a -> Bool`
  - More about type classes (including defining your own) next time.

# Working with lists

- Have already seen how to take apart *tuples*
  - `add (x, y) = x + y`
  - ``let (q, r) = 10 `quotRem` 3 in ...``
- For lists, note that `[3,4,5]` syntax is actually *syntactic sugar* for `3 : (4 : (5 : []))`
  - `[] :: [a]` is sometimes called *nil*.
  - `(:) :: a -> [a] -> [a]` is usually called *cons*.
    - Beware: Haskell swaps meanings of `:` and `::` compared to ML/F#!
- Any well-formed list (and there is no other kind!) is either empty (`[]`) or of the form ($h$ : $t$) for some $h$ and $t$.
- Can define functions over lists by covering both possibilities:

```
myReverse :: [a] -> [a]
myReverse [] = []
myReverse (h : t) = myReverse t ++ [h]
```

# Pattern matching, continued

- Can pattern-match on several arguments at once:

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x <= y then x : merge xs (y:ys)
                                 else y : merge (x:xs) ys
```

- In case of overlaps, *first* successful match is chosen.
- `ghci -W` *warns* about uncovered cases.
  - Runtime error if matching fails
- Can also use `case`-expressions for pattern matching:

```
case filter isOK attempts of
  [] -> "no solutions"
  [_x] -> "one solution" -- "_x" suppresses a warning
  _ -> "several solutions"
```

(Again, indentation is significant; or separate branches with "`;`".)

- Wildcard pattern _ matches everything

# Even more pattern matching

- Patterns must only bind variables at most once; this is **illegal**:

```
myElem :: a -> [a] -> Bool
myElem x [] = False
myElem x (x:ys) = True  -- note: x occurs twice
myElem x (_:ys) = myElem x ys
```

- But can write with explicit Boolean *guard* on pattern:

```
myElem :: Eq a => a -> [a] -> Bool
myElem x [] = False
myElem x (y:ys) | x == y = True
myElem x (_:ys) = myElem x ys
```

- If guard evaluates to `False`, matching resumes with next case.

# Programmer-defined data types

- Most non-trivial Haskell programs contain problem-specific type definitions.
- Simplest kind: *type synonyms* (abbreviations)

```
type Name = (String, String)  -- family & given name
("Wegener", "Henrik") :: Name
```

- Types may be *enumerations*:

```
data Color = Red | Green | Blue
  deriving (Show, Eq)
```

  The `deriving` clause adds `Color` to the respective type classes.
- **Note:** both type name and *constructor* names must start with uppercase letter.
- Actually, `Bool` is just a predefined enumeration:

```
data Bool = False | True
  deriving (Show, Eq, ...)
```

# Value-carrying constructors

- Can associate extra data with some or all constructors:

```haskell
data Figure = Point
            | Disc Double -- radius
            | Rectangle Double Double -- width, height

myFigure = Rectangle 3.0 4.0
```

- Define functions on datatype by pattern matching:

```haskell
area :: Figure -> Double
area Point = 0.0
area (Disc r) = pi * r ^ 2
area (Rectangle w h) = w * h
```

(Note parentheses around non-atomic patterns)

# Record notation

- Sometimes not obvious what constructor arguments represent.
  - Simple solution: comments
- Alternative: *named fields*

```
data Figure = Point
            | Disc {radius :: Double}
            | Rectangle {width, height :: Double}
```

- Can use either positional or named style when constructing:

```
myFigure = Rectangle 3.0 4.0
myFigure = Rectangle {height = 4.0, width = 3.0}
```

- Can use field names to *project* out components

```
let a = width fig * height fig in ...
```

  - **Note:** signals runtime error if `fig` is not a `Rectangle`
    - So normally use projections only for datatypes with exactly one constructor

# More datatypes

- Datatype definitions may be *recursive*:

```
data Figure = ...
            | Stack Figure Figure
```

- Then functions on them are normally also recursive:

```
...
area (Stack f1 f2) = area f1 + area f2
```

- Datatypes may be polymorphic:

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

myTree :: Tree Int
myTree = Node (Leaf 2) (Node (Leaf 3) (Leaf 4))
```

- Mutual recursion, possibly mixing `type` and `data` definitions:

```
data RoseTree a = RoseTree a (Forest a)  -- data, children
type Forest a = [RoseTree a]  -- zero or more trees
```

# A few more built-in datatypes

- Have already seen lists:

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

  **Note:** infix *constructors* start with colon
  - ... which is why infix *operators* must not.
  - Always possible to tell visually whether a name occurring in pattern is a constructor or a variable.

- Option (or "nullable") types

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

  Useful especially for function return types:

  ```
  lookup :: Eq a => a -> [(a, b)] -> Maybe b
  ```

- Disjoint-union types:

  ```
  data Either a b = Left a | Right b
  ```

  So type `Maybe` *a* works almost the same as `Either ()` *a*.
  - `()` is the empty-tuple type, containing a single value `()`.

# Types vs. sets (cf. Quiz 1 question 8)

- Every Haskell type $a$ conceptually denotes a mathematical set $\mathcal{S}(a)$ of values:

$$
\begin{aligned}
\mathcal{S}(\texttt{Bool}) &= \{\mathit{False}, \mathit{True}\} \\
\mathcal{S}(\texttt{Integer}) &= \mathbb{Z} \quad \text{(while } \mathcal{S}(\texttt{Int}) = \mathbb{Z}/2^{64} \text{ on most platforms)} \\
\mathcal{S}(\texttt{(}a\texttt{,}b\texttt{)}) &= \mathcal{S}(a) \times \mathcal{S}(b) = \{(x, y) \mid x \in \mathcal{S}(a) \wedge y \in \mathcal{S}(b)\} \\
\mathcal{S}(\texttt{Either } a\ b) &= \mathcal{S}(a) \uplus \mathcal{S}(b) = \{(1, x) \mid x \in \mathcal{S}(a)\} \cup \{(2, y) \mid y \in \mathcal{S}(b)\} \\
\mathcal{S}(a \texttt{ -> } b) &= \mathcal{S}(b)^{\mathcal{S}(a)} = \\
&\qquad \{F \subseteq \mathcal{S}(a) \times \mathcal{S}(b) \mid \forall x \in \mathcal{S}(a).\, \exists! y \in \mathcal{S}(b).\, (a, b) \in F\}
\end{aligned}
$$

- Write $|X|$ for *cardinality* (# of elements) of finite set $X$. Then $|A \times B| = |A| \cdot |B|$, $|A \uplus B| = |A| + |B|$, $|B^A| = |B|^{|A|}$.
- Particularly useful to keep in mind for (possibly unfamiliar) `Either` type constructor.
- In Haskell, this correspondence is slightly complicated by addition of *undefined* elements for most types.
  - E.g., actually $\mathcal{S}(\texttt{Bool}) = \{\mathit{False}, \mathit{True}, \bot\}$
  - Due to lazy evaluation (next time)

# Tasks for this week

- Install Haskell on your computer
  - See Absalon page for details; get help if needed
- Attend exercise classes/labs 10–11 or 11–12
  - See **today's** room assignments on Absalon
- Talk to fellow students about forming assignment/study groups
  - Study groups are typically size 4, assignment groups are max 2.
  - Fill out "My study group" questionnaire if you want help finding a study group (which can be split up for the assignments)
- Work on Exercise Set 1
  - *Start* at lab sessions today
- Use discussion fora on Absalon/Discord for questions outside of lecture and lab hours
  - Please open new (named) discussion thread for each topic on Absalon
- Attend Lecture 2 and labs on Thursday (go to correct rooms!)
- Assignment 1 will be out Thursday, due Friday of next week.