This Lecture (1)
Overview of Haskell. Not necessarily very systematic, but I hope to:

• Review some concepts and ideas from Part I in the setting of Haskell
• Give you a good idea what Haskell looks like
• Make you aware of central features
• Highlight some differences to SML/OCaml
• Point out some common pitfalls

You’ll get a chance to hone your Haskell skills in a lab session after this lecture.

What Is a Functional Language? (2)
This “definition” covers both:

• **Pure** functional languages: no side effects
  - (Weakly) declarative: equational reasoning valid (with care); referentially transparent.
  - Example: Haskell
• **Mostly** functional languages: some side effects, e.g. for I/O.
  - Equational reasoning valid for pure fragments.
  - Examples: ML, OCaml, Scheme, Erlang

(Real purists would point out that non-termination can be seen as a side effect.)
Example: Computing Sums (1)

Summing the integers from 1 to 10000 in Java:

```java
total = 0;
for (i = 1; i <= 10000; ++i)
    total = total + 1;
```

The method of computation is to execute operations in sequence, in particular variable assignment.

Example: Computing Sums (2)

Summing the integers from 1 to 10000 in the functional language Haskell:

```haskell
sum [1..10000]
```

The method of computation is function application.

Of course, essentially the same program could be written in, say, Java. Does that make Java a functional language? Discuss!

Example: Computing Sums (3)

Some reasons not to adopt the “functional approach” in Java:

- Syntactically awkward (even given suitable library definitions)
- Temporarily creating a list of 10000 integers just to add them seems highly objectionable; not good Java style.

But isn’t the second point a good argument against the “functional approach” in general?

Example: Computing Sums (4)

Actually, no!

- Nothing says the entire list needs to be created at once. In lazy languages, like Haskell, the list will be generated as needed, element by element.
- Nothing says the list needs to be created at all! Compilers for functional languages, thanks to equational reasoning being valid, are often able to completely eliminate intermediate data structures.
Example: Computing Sums (5)

- Note that the Haskell code is *modular*, while the Java code is not.
- Being overly prescriptive regarding computational details (evaluation order) often hampers modularity.

We will discuss the last point in more depth later.

Typical Functional Features (1)

Nevertheless, some typical features and characteristics of functional languages can be identified:

- Light-weight notation geared at
  - defining functions
  - expressing computation through function application.
- Functions are first-class entities.
- Recursive (and co-recursive) function and data definitions central.

Typical Functional Features (2)

- Implementation techniques aimed at executing code expressed in a functional style efficiently.

More?

This and the Following Lectures

- In this and the following lectures we will explore *Purely Functional Programming* through the use of *Haskell*.
- Some themes:
  - Relinquishing control: exploiting lazy evaluation
  - Purely functional data structures
  - Effects without compromising purity
  - Concurrency in a pure FP setting
  - Haskell features (e.g. Type Classes)
The GHC System (1)

- GHC supports Haskell 98, Haskell 2010, and many extensions
- GHC is currently the most advanced Haskell system available
- GHC is a compiler, but can also be used interactively: ideal for serious development as well as teaching and prototyping purposes

The GHC System (2)

On a Unix system, GHCi can be started from the prompt by simply typing the command ghci:

```
sis-1% ghci
[
 ___ ___ _
/ _ \/ _\ /\ /\ ___(_)
/ /_\// /_/ /_/ | | GHC Interactive, version 6.3, for Haskell 98.
/ /\\ _ / /\_/ | | http://www.haskell.org/ghc/
\____/\ /\____/\|_ Type :? for help.
Loading package base ... linking ... done.
Prelude>
```

The GHC System (3)

The GHCi > prompt means that the GHCi system is ready to evaluate an expression. For example:

```
> 2+3*4
14
> reverse [1,2,3]
[3,2,1]
> take 3 [1,2,3,4,5]
[1,2,3]
```

Function Application (1)

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

```
f(a,b) + c \ d
```

“Apply the function \( f \) to \( a \) and \( b \), and add the result to the product of \( c \) and \( d \).”
In Haskell, *function application* is denoted using `space`, and multiplication is denoted using `*`.

```
   f a b + c*d  
```

Meaning as before, but Haskell syntax.

Moreover, function application is assumed to have *higher priority* than all other operators. For example:

```
   f a + b  
```

means

```
   (f a) + b  
```

not

```
   f (a + b)  
```

**What is a Type?**

A *type* is a name for a collection of related values. For example, in Haskell the basic type `Bool` contains the two logical values `False` and `True`.

**Types in Haskell**

- If evaluating an expression `e` would produce a value of type `t`, then `e` has type `t`, written `e :: t`.

- Every well-formed expression has a type. It can usually be calculated automatically at compile time using a process called *type inference* or *type reconstruction* (Hindley-Milner).

- However, giving manifest type declarations for at least top-level definitions is good practice.

- Sometimes necessary to state type explicitly, e.g. polymorphic recursion.
Basic Types

Haskell has a number of basic types, including:

- **Bool**: Logical values
- **Char**: Single characters
- **Int**: Fixed-precision integers
- **Integer**: Arbitrary-precision integers
- **Double**: Double-precision floating point

List Types (1)

A list is a sequence of values of the same type:

- \([\text{False, True, False}] :: \text{[Bool]}\)
- \([\text{’a’, ’b’, ’c’, ’d’}] :: \text{[Char]}\)

In general:

- \([t]::\text{[type]}\)

List Types (2)

Haskell defines the string string type to be a list of characters:

- \(\text{type String = [Char]}\)

String syntax is supported. For example:

- \(\text{“abcd” = [’a’, ’b’, ’c’, ’d’]}\)

Tuple Types

A tuple is a sequence of values of different types:

- \((\text{False, True}) :: (\text{Bool, Bool})\)
- \((\text{False, ’a’, True}) :: (\text{Bool, Char, Bool})\)

In general:

- \((t_1, t_2, \ldots, t_n)\) is the type of \(n\)-tuples whose \(i^{th}\) component has type \(t_i\) for \(i \in [1 \ldots n]\).
Aside: Naming Conventions

Haskell *enforces* certain naming conventions. For example:

- Type constructors (like `Bool`) and value constructors (like `True`) always begin with a capital letter.
- Variables (including function names) always begin with a lowercase letter.

A somewhat similar convention applies to infix operators where constructors are distinguished by starting with a colon (`:`).

Function Types (1)

A *function* is a mapping from values of one type to values of another type:

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool}
\]

In general:

\[
t_1 \rightarrow t_2 \text{ is the type of functions that map values of type } t_1 \text{ to values of type } t_2.
\]

Function Types (2)

If a function needs more than one argument, pass a tuple, or use *Currying*:

\[
(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

This really means:

\[
(\&\&) :: \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})
\]

Idea: arguments are applied one by one. This allows *partial application*.

Aside: Functions and Operators

- Any (infix) operator can be used as a (prefix) function by enclosing it in parentheses. E.g.:
  \[
  \text{True} \&\& \text{False}
  \]
  is equivalent to
  \[
  (\&\&) \text{True False}
  \]

- Any function can be used as an operator by enclosing it in back quotes. E.g.:
  \[
  \text{add} 1 2
  \]
  is equivalent to
  \[
  1 \text{`add`} 2
  \]
Polymorphic Functions (1)

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

\[ \text{length} :: [a] \rightarrow \text{Int} \]

"For any type \( a \), length takes a list of values of type \( a \) and returns an integer."

This is called **Parametric Polymorphism**.

Polymorphic Functions (2)

The type signature of length is really:

\[ \text{length} :: \forall a . \ [a] \rightarrow \text{Int} \]

- It is understood that \( a \) is a type variable, and thus it ranges over all possible types.
- Haskell 98 does not allow explicit foralls: all type variables are implicitly qualified at the outermost level.
- Haskell extensions allow explicit foralls.

Exercise

Given:

\[ \text{id} :: a \rightarrow a \]
\[ \text{not} :: \text{Bool} \rightarrow \text{Bool} \]
\[ \text{foo} :: (a \rightarrow a) \rightarrow a \rightarrow a \]
\[ \text{fie} :: (\forall a . \ a \rightarrow a) \rightarrow a \rightarrow a \]

what is the type of each of:

\[ \text{foo id} :: ?? \]
\[ \text{foo not} :: ?? \]
\[ \text{fie id} :: ?? \]
\[ \text{fie not} :: ?? \]

Types are Central in Haskell

Types in Haskell play a much more central role than in many other languages. Some reasons:

- Haskell's type system is very expressive thanks to Parametric Polymorphism:
  \[ (++) :: [a] \rightarrow [a] \rightarrow [a] \]

- The types say a lot about what functions do because Haskell is a pure language: no side effects (Referential Transparency).

For example, all a function of type \( \text{Int} \rightarrow \text{Int} \) can do is to return an integer or fail to terminate. Cannot launch a missile behind our backs.
**Parametricity**

In fact, due to a property called **parametricity**, it goes even further: polymorphic types give rise to **free theorems** (Wadler 1989). For example:

For any function \( r :: \forall a . [a] \rightarrow [a] \), and every total function \( f :: t_1 \rightarrow t_2 \) for some specific types \( t_1 \) and \( t_2 \), we have:

\[
\text{map } f \cdot r = r \cdot \text{map } f
\]

This holds by virtue of \( r \)'s polymorphic type: no need to even consider its definition!

---

**Conditional Expressions**

As in most programming languages, functions can be defined using **conditional expressions**:

\[
\text{abs} :: \text{Int} \rightarrow \text{Int}
\]

\[
\text{abs } n = \text{if } n \geq 0 \text{ then } n \text{ else } -n
\]

Alternatively, such a function can be defined using **guards**:

\[
\text{abs} :: \text{Int} \rightarrow \text{Int}
\]

\[
\text{abs } n \mid n \geq 0 \quad = n
\]

\[
\mid \text{otherwise} = -n
\]

---

**Hoogle**

Hoogle is a Haskell API search engine:

http://www.haskell.org/hoogle/

Allows searching by function name or by **approximate type signature**.

For example, searching on

\[
(a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

turns up \text{map}, \text{fmap}, ...
Case expressions allow pattern matching to be performed wherever an expression is allowed, not just at the top-level of a function definition:

```haskell
not :: Bool -> Bool
not b = case b of
  False -> True
  True  -> False
```

Haskell uses layout (indentation) to group code into blocks. For example, the following is a syntax error:

```haskell
not b = case b of
  False -> True
  True  -> False
```

Alternatively, explicit braces and semicolons can be used. It's even possible to mix and match:

```haskell
not b = case b of
  { False -> True ;
    True  -> False }
```

List Patterns (1)

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, starting from [], the empty list.

Thus:

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

means

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

List patterns (2)

Functions on lists can be defined using $x:xs$ patterns:

```haskell
head :: [a] -> a
head (x:_  ) = x

tail :: [a] -> [a]
tail ( _:xs) = xs
```
Pattern Matching and Guards

Pattern matching and guards may be combined:

\[
dropWhile :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
dropWhile _ [] = []
dropWhile p xxs@(x:xs)
  | p x      = dropWhile p xs
  | otherwise = xxs
\]

List Comprehensions

List comprehensions, similar to standard mathematical set notation, are very useful for expressing computations on lists:

\[
[ x \times x \mid x \leftarrow [1..10], \text{odd } x ]
= [1,9,25,49,81]
\]

\[
[ (x,y) \mid x \leftarrow [1..10],
  y \leftarrow [1..10],
  \text{even } (x + y) ]
= [(1,1),(1,3),(1,5),... (10,8),(10,10)]
\]

Lambda Expressions

A function can be constructed without giving it a name by using a lambda expression:

\[
\x \rightarrow x + 1
\]

“The nameless function that takes a number \( x \) and returns the result \( x + 1 \)”

Note that the ASCII character \( \\backslash \) stands for \( \lambda \) (lambda).

Currying Revisited

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

\[
\text{add } x \ y = x + y
\]

means

\[
\text{add } = \x \rightarrow (\y \rightarrow x + y)
\]
Aside: Operator Sections

Another syntactic nicety in Haskell is partially applied operators or operator sections. For example:

\[ (+1) = \lambda x \rightarrow x + 1 \]  \ Add 1
\[ (1+) = \lambda x \rightarrow 1 + x \]  \ Add 1
\[ (*2) = \lambda x \rightarrow x * 2 \]  \ Multiply by 2
\[ (/2) = \lambda x \rightarrow x / 2 \]  \ Divide by 2
\[ (1/) = \lambda x \rightarrow 1 / x \]  \ Reciprocal

Recursive Definitions

- Definitions in Haskell may in general be (mutually) recursive.
- No special letrec form.
- Order of definition is immaterial.

\[
\text{foo } x = \ldots \text{ fum } (x - 1) \\
\text{fie } x = \ldots \text{ fie } (x - 1) \\
\text{fum } x = \ldots \text{ foo } (x - 1) \\
\]

- To allow inference of maximally polymorphic types, definitions are grouped into minimal recursive groups prior to type checking.

Local Definitions

Haskell provides two ways to introduce local definitions:

- let-expressions
- where-clauses

\[
\text{f } x = \text{h } x + c \\
\text{g } x = \text{let} \\
\text{where} \\
\text{h } x = x * x \\
c = 100 \\
in \\
h x + c
\]

Again, the definitions can be (mutually) recursive.

Data Declarations (1)

A new type can be declared by specifying its set of values using a data declaration. For example, Bool is in principle defined as:

```
data Bool = False | True
```
Data Declarations (2)

What happens is:

- A new type `Bool` is introduced
- **Constructors** (functions to build values of the type) are introduced:
  
  ```
  False :: Bool
  True :: Bool
  ```
  
  (In this case, just constants.)
- Since constructor functions are bijective, and thus in particular injective, pattern matching can be used to take apart values of defined types.

Recursive Types (1)

In Haskell, new types can be declared in terms of themselves. That is, types can be **recursive**:

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

`Nat` is a new type with constructors

- `Zero :: Nat`
- `Succ :: Nat -> Nat`

Effectively, we get both a new way form terms and typing rules for these new terms.

Data Declarations (3)

Values of new types can be used in the same ways as those of built in types. E.g., given:

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

we can define:

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

flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```

Recursive Types (2)

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)
```
Recursion and Recursive Types

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

```haskell
nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n | n >= 1 = Succ (int2nat (n - 1))
```

Parameterized Types

Types can also be parameterized on other types:

```haskell
data List a = Nil | Cons a (List a)

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

Resulting constructors:

```haskell
Nil :: List a
Cons :: a -> List a -> List a
Leaf :: a -> Tree a
Node :: Tree a -> Tree a -> Tree a
```

Overloading (1)

Haskell supports a form of overloading: using the same name to refer to different definitions depending on the involved types. For example:

```haskell
(==) :: Eq a => a -> a -> Bool
```

This means `==' is defined for any type `a` belonging to the type class `Eq`.

Overloading (2)

In particular, `Bool` and `Char` both belong to `Eq`, so the following two expressions are well-typed:

```haskell
True == False
'a' == 'b'
```

Behind the scenes, the equality test is dispatched to the appropriate function for `Bool` and `Eq` respectively.
Overloading (3)

We will discuss type classes in more depth later. However, it is useful to know that Haskell allows class instances for new types to be derived for a handful of built-in classes, notably `Eq`, `Ord`, and `Show`:

```haskell
data Nat = Zero | Succ Nat deriving (Eq, Ord, Show)
```

Now `show (Succ (Succ Zero))` yields "Succ (Succ Zero)."

Modules in Haskell (1)

- A Haskell program consists of a set of modules.
- A module contains definitions:
  - functions
  - types
  - type classes
- The top module is called `Main`:

```haskell
module Main where
main = putStrLn "Hello World!"
```

Modules in Haskell (2)

By default, only entities defined within a module are in scope. But a module can import other modules, bringing their definitions into scope:

```haskell
module A where
f1 x = x + x
f2 x = x + 3
f3 x = 7

module B where
import A
g x = f1 x * f2 x + f3 x
```

The Prelude

There is one special module called the Prelude. It is imported implicitly into every module and contains standard definitions, e.g.:

- Basic types (`Int`, `Bool`, `tuples`, `[]`, `Maybe`, …)
- Basic arithmetic operations (`+`, `*`, …)
- Basic tuple and list operations (`fst`, `snd`, `head`, `tail`, `take`, `map`, `filter`, `length`, `zip`, `unzip`, …)

(It is possible to explicitly exclude (parts of) the Prelude if necessary.)
Qualified Names (1)

The **fully qualified name** of an entity \(x\) defined in module \(M\) is \(M.x\).

\[
g \ x = A.f1 \ x \ * \ A.f2 \ x + f3 \ x
\]

**Note! Different from function composition!!!**

Always write function composition with spaces:

\[
f \ . \ g
\]

The module **name space** is **hierarchical**, with names of the form \(M_1.M_2.\ldots.M_n\). This allows related modules to be grouped together.

Import Variations

Another way to resolve name clashes is to be more precise about imports:

- `import A (f1,f2)`
- `import A hiding (f1,f2)`
- `import qualified A`

Can be combined in all possible ways; e.g.:

- `import qualified A hiding (f1, f2)`

Qualified Names (2)

Fully qualified names can be used to resolve name clashes. Consider:

```plaintext
module A where
module C where
  f x = 2 * x
import A
import B
module B where
  f x = 3 * x
g x = A.f x + B.f x
```

Two different functions with the same unqualified name \(f\) in scope in \(C\). Need to write \(A.f\) or \(B.f\) to disambiguate.

Export Lists

It is also possible to be precise about what is **exported**:

```
module A (f1, f2) where
  ...
```

Various abbreviations possible; e.g.:
- A type constructor along with all its value constructors
- Everything imported from a specific module
Suppose we need to represent data about people:

- Name
- Age
- Phone number
- Post code

One possibility: use a tuple:

```haskell
type Person = (String, Int, String, String)
henrik = ("Henrik", 25, "8466506", "NG92YZ")
```

Problems? Well, the type does not say much about the purpose of the fields! Easy to make mistakes; e.g.:

```haskell
def getPhoneNumber :: Person -> String
getPhoneNumber (_, _, _, pn) = pn
```

or

```haskell
henrik = ("Henrik", 25, "NG92YZ", "8466506")
```

Can we do better? Yes, we can introduce a new type with named fields:

```haskell
data Person = Person
    { name :: String,
      age :: Int,
      phone :: String,
      postcode :: String
    }

deriving (Eq, Show)
```

Labelled fields are just “syntactic sugar”: the defined type really is this:

```haskell
data Person = Person String Int String String
```

and can be used as normal.

However, additionally, the field names can be used to facilitate:

- Construction
- Update
- Selection
- Pattern matching
**Construction**

We can construct data without having to remember the field order:

```plaintext
henrik = Person {
  age = 25,
  name = "Henrik",
  postcode = "NG92YZ",
  phone = "8466506"
}
```

**Update (1)**

Fields can be “updated”, creating new values from old:

```plaintext
> henrik { phone = "1234567" } = Person {name = "Henrik", age = 25,
phone = "1234567", postcode = "NG92YZ"}
```

Note: This is a functional “update”! The old value is left intact.

**Update (2)**

How does “update” work?

```plaintext
henrik { phone = "1234567" } = f (Person a1 a2 _ a4)
   Person a1 a2 "1234567" a4
f henrik
```

**Selection**

We automatically get a selector function for each field:

```plaintext
name :: Person -> String
age :: Person -> Int
phone :: Person -> String
postcode :: Person -> String
```

For example:

```plaintext
> name henrik
"Henrik"
> phone henrik
"8466506"
```
Pattern matching

Field names can be used in pattern matching, allowing us to forget about the field order and pick only fields of interest.

\[
\text{phoneAge (Person \{phone = p, age = a\}) = p ++":" ++ show a}
\]

This facilitates adding new fields to a type as most of the pattern matching code usually can be left unchanged.

Multiple Value Constructors (1)

\[
\text{data Being = Person \{name :: String, age :: Int, phone :: String, postcode :: String \}}
\]

\[
| \text{Alien \{name :: String, age :: Int, homeland :: String \}}
\]

\[
\text{deriving (Eq, Show)}
\]

Multiple Value Constructors (2)

It is OK to have the same field labels for different constructors as long as their types agree.

Distinct Field Labels for Distinct Types

It is not possible to have the same field names for different types! The following does not work:

\[
\text{data X = MkX \{field1 :: Int\}}
\]

\[
\text{data Y = MkY \{field1 :: Int, field2 :: Int\}}
\]

One work-around: use a prefix convention:

\[
\text{data X = MkX \{xField1 :: Int\}}
\]

\[
\text{data Y = MkY \{yField1 :: Int, yField2 :: Int\}}
\]
Advantages of Labelled Fields

- Makes intent clearer.
- Allows construction and pattern matching without having to remember the field order.
- Provides a convenient update notation.
- Allows to focus on specific fields of interest when pattern matching.
- Addition or removal of fields only affects function definitions where these fields really are used.

Reading

- Philip Wadler. Theorems for Free! In *Functional Programming Languages and Computer Architecture, FPCA’89*, 1989