



# CS 115

# Functional Programming

*Lecture 2:*  
**Evaluation**



*Functional Programming*



# Today

- More Haskell basics
- Introduction to Haskell's evaluation model





# More Haskell basics





# Scalar data types

- Haskell has a fairly standard assortment of scalar data types:
  - `Int`
  - `Integer`
  - `Float`
  - `Double`
  - `Char`
  - `Bool`





# Int and Integer

- The two basic integral types are **Int** and **Integer**
- **Int** stands for machine-level integers (32 or 64 bits, as the case may be)
- **Integer** stands for arbitrary-precision integers
- If there is no compelling reason to use **Int**, use **Integer**
- (Other integral types also exist)





# Float and Double

- The two basic approximate real number types are **Float** and **Double**
- Both map onto corresponding machine types (IEEE Floats, IEEE Doubles)





# Char

- The **Char** type represents a single Unicode character (**ghc** uses UTF-8 encoding)
- Character literals written between single quotes  
`'l' 'i' 'k' 'e' ' ' 't' 'h' 'i' 's'`
- Standard character escape sequences like `\n` (newline), `\t` (tab) and `\\"` (backslash) are supported





# Bool

- The **Bool** type represents boolean (true/false) values
- There are two values in the **Bool** type: **True** and **False**
- These are actually data constructors, and the **Bool** type is not hard-wired into the language
  - instead, just part of the **Prelude** (core libraries)





# Compound data types

- Haskell has a number of built-in compound data types
  - meaning made of multiple instances of simpler data types
- Examples:
  - lists
  - strings
  - tuples
- Also many other compound types in libraries
  - arrays, sets, maps, etc.





# Lists

- Lists in Haskell are comprised of multiple values of a single type (list of `Int`, list of `Float`, etc.)
- Literal lists are written with values between square brackets, separated by commas

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

- Type of lists is written as the type of the elements, surrounded by square brackets

`[1, 2, 3, 4, 5] :: [Integer]`





# Lists

- Empty list written as `[]`
- Lists are constructed using the `:` (cons) operator

```
1 : (2 : (3 : (4 : (5 : []))))  
== [1, 2, 3, 4, 5]
```

- We will examine lists in much greater detail next lecture





# Strings

- Strings are represented as lists of **Chars**
- Advantage: can use all the list functions on strings
- Disadvantage: this is a very expensive way to represent strings!
  - alternatives are available e.g. **ByteString** and **Text**
- There is a data type called **String** which is an alias for **[Char]** (list of **Char**)
- Literal strings written between double quotes **"like this"**; usual escapes apply





# Tuples

- A tuple is a sequence of values inside parentheses, separated by commas
- Tuples can contain values of different types:  
`(1, "foo", 3.14) :: (Integer, String, Double)`
- There are no empty tuples or length-1 tuples
- Can construct tuples using "tuple constructors"
  - `(, ) 1 2 → (1, 2)`
  - `(, , ) 1 "foo" 3.14 → (1, "foo", 3.14)`
- but usually just write out literal tuples





# Identifier syntax

- Haskell has fairly conventional syntax for identifiers
  - letters from **a-z, A-Z**, numbers from **0-9**, also **\_**
  - also **'** character allowed e.g. **foo'**, **foo''**
  - first character cannot be digit or **'**
  - first character must be capitalized in some circumstances:
    - type names (**Int, Integer, Float, Char**)
    - module names (**Prelude, Data.List**)
    - data constructor names (later lecture)
    - type constructor names (later lecture)
    - nowhere else!





# Operator syntax

- There is no fixed set of operators
  - operators are not "hard-wired" into the language
- Operator identifiers are made up of "operator characters" (usual symbolic characters on keyboard)
- Operators are just syntactic sugar for two-argument functions in infix position
- Can convert an operator to a two-argument function by surrounding it with parentheses

(+) 2 2 → 4





# Function → Operator

- A two-argument function can be written as an operator by surrounding it with backticks (the ` character)
- Example:

```
Prelude> mod 5 2
```

```
1
```

```
Prelude> 5 `mod` 2
```

```
1
```

- This often makes code more readable





# Defining new operators

- Operators can be defined as easily as functions:

```
(%%) :: Integer -> Integer -> Integer
```

```
(%%) x y = x + 2 * y
```

or write second line as:

```
x %% y = x + 2 * y
```

- Test:

```
Prelude> 10 %% 2
```

```
14
```

- Cool!





# Operator precedence

- Haskell operators have one of ten precedence levels (0-9); 0 is lowest, 9 is highest
- Function application has higher precedence than anything else (conceptually, level 10)
  - `double 10 + double 20` → `(double 10) + (double 20)`
- Can use `ghci`'s `:info (:i)` command to tell you what the precedence for an operator is





# Operator precedence

```
Prelude> :info *
class Num a where
...
(*) :: a -> a -> a
...
-- Defined in GHC.Num
```

**infixl 7 \***

- We'll explain the **class** stuff later
- **infixl 7 \*** means the **\*** operator is left-associative, precedence level 7





# Operator precedence

```
Prelude> :info +
```

```
infixl 6 +
```

- `+` has lower precedence than `*`

```
Prelude> :i ^
```

```
infixr 8 ^
```

- `^` (exponentiation) has higher precedence than `*`, right associative (so `a^b^c` → `a^(b^c)`)
- N.B. `^` operator used when raising to integer power only; use `**` for raising to float power (also `infixr 8`)





# Operator precedence

- When defining a new operator, default precedence is 9 (highest); default associativity is left
- Can specify precedence/associativity explicitly

```
(%%) :: Integer -> Integer -> Integer
```

```
x %% y = x + 2 * y
```

```
infixl 7 %%
```

- **infixl** → left associative
- **infixr** → right associative
- **infix** → non-associative





# Functions with multiple arguments

- Conceptually, Haskell functions all take only a single argument
- We need to be able to write functions that take multiple arguments
- Two basic ways to do this





# Functions with multiple arguments

- Way 1:

```
add :: (Integer, Integer) -> Integer
```

```
add (x, y) = x + y
```

- The **add** function takes as its only argument a two-tuple of **Integers**, returning an **Integer**
- The left-hand side of the equation pattern matches the two-tuple, binding **x** and **y** locally in the equation (scope includes the right-hand side of the equation only)
- Call this function like this: **add (3, 2) → 5**





# Functions with multiple arguments

- Way 2:

```
add2 :: Integer -> Integer -> Integer  
add2 x y = x + y
```

- The `add2` function takes as its only argument a single `Integer`, returning a value with the functional type `Integer -> Integer`
- N.B. The function arrow `->` associates to the right, so the type signature is really `Integer -> (Integer -> Integer)`





# Functions with multiple arguments

- Way 2:

`add2 :: Integer -> Integer -> Integer`

`add2 x y = x + y`

- Calling this function:

`add2 3 4 → 7`

- Function calls associate to the left, so this is really `((add2 3) 4)`
- What does `(add2 3)` mean?





# Functions with multiple arguments

- Can use partially-applied functions as functions:

```
add_3 :: Integer -> Integer  
add_3 = add2 3
```

```
Prelude> add_3 10  
13
```

- This behavior is called "currying"
  - after Haskell Curry, a logician
  - (also inspired a programming language...)





# Functions with multiple arguments

- Pitfall:

```
square :: Integer -> Integer
```

```
square x = x * x
```

```
Prelude> square square 4
```

- Get nasty error message
- Haskell interprets this as **(square square) 4** which doesn't make sense
- Recall: function application associates to the left!
- Need to write **square (square 4)**





# Operator sections

- Can do the equivalent of currying on operators too:

```
Prelude> (*2) 10
```

```
20
```

```
Prelude> (9/) 3
```

```
3
```

- These are called "operator sections"

```
squared :: Integer -> Integer
```

```
squared = (^2)
```





# Local definitions

- Often useful to have local definitions in functions:
  - local values: compute once, use multiple times
  - local functions: use only in the scope of the outer function
- Two ways to do this in Haskell:
  - **let** expressions
  - **where** declarations





# let expression

- **let** expression defines a local value or values and a scope to use it in

```
let x = 10 in 2 * x
```

→ 20

- Can define multiple local values in a single **let**

```
let x = 10  
      y = 100
```

```
in x - y
```

→ -90





# let expression

- Names in a `let` expression can depend on each other:

```
let x = 10  
    y = x * 2
```

```
in x + y
```

→ 30

```
let y = x * 2  
    x = 10
```

```
in x + y
```

→ 30





# let expression

- Local definitions in a **let** expression can be functions (even recursive functions):

```
let f x = 2 * x in f 1000
```

→ 2000

```
let
```

```
  odd n = if n == 0 then False else even (n-1)
```

```
  even n = if n == 0 then True else odd (n-1)
```

```
in even 1002
```

→ True





# let expression

- Can even add type signatures to local functions:

`let`

`odd :: Integer -> Integer`

`odd n = if n == 0 then False else even (n-1)`

`even :: Integer -> Integer`

`even n = if n == 0 then True else odd (n-1)`

`in even 1002`

`→ True`

- This is recommended!





# where declaration

- After a function equation in a function definition, can add a **where** declaration for definitions local to that equation

```
-- tail-recursive factorial
factorial_tr :: Integer -> Integer
factorial_tr n = iter n 1
  where
    iter :: Integer -> Integer -> Integer
    iter 0 r = r
    iter n r = iter (n - 1) (n * r)
```





# where declaration

- **where** declaration is not an expression
  - can't write `(x * 2 where x = 100^2)`
- Scope of **where** is only the equation to which it applies
  - won't apply to multiple equations in the same function
- Can add type signature to names bound in a **where** declaration
  - not required (types inferred if not supplied) but almost always a good idea
- **where** generally preferred over **let** for local function definitions





# Haskell's evaluation model





# Evaluation in Haskell

- The "evaluation model" of a language are the rules by which expressions get evaluated
- Good news: Haskell's evaluation model is generally very simple
  - no more than high school algebra
  - "equational reasoning"
- Bad news: lazy evaluation complicates things significantly in some cases
- Let's walk through some examples





# Example 1

```
double :: Integer -> Integer
```

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

- Evaluate: **double (3 \* 4)**
- Multiple possibilities exist!
- In general:
  - pick a *reducible expression* (**redex**) and reduce it
  - continue until there is nothing more to reduce
  - the resulting value is called the *normal form*
  - which is the answer





# Example I

- Evaluate: **double (3 \* 4)**
- Attempt 1:
  - reduce **(3 \* 4)** first → **12**
  - evaluate **double 12**
  - replace **double** by its definition, substitute values for arguments
  - evaluate **12 + 12** → **24**
- This strategy is called *strict* or *applicative-order* evaluation
- You first reduce arguments to functions to normal forms, then substitute into function body





# Example 1

- Evaluate: **double (3 \* 4)**
- Attempt 2:
  - replace **double** by its definition, substitute unevaluated expressions for arguments
  - evaluate **(3 \* 4) + (3 \* 4)**
  - reduce left subexpression → **12 + (3 \* 4)**
  - reduce right subexpression → **12 + 12**
  - reduce remaining expression → **24**
- This is called *non-strict* or *normal-order* evaluation
- Apply functions to unevaluated expressions, reduce only as needed to get final result





# Example 1

- Evaluate: **double (3 \* 4)**
- Attempt 3:
  - replace **double** by its definition, substitute unevaluated expressions for arguments
  - evaluate **(3 \* 4) + (3 \* 4)**
  - both **(3 \* 4)** subexpressions are actually the same expression, so reduce them both at the same time
  - → **12 + 12 → 24**
- This is usually called *lazy evaluation*
  - Optimized form of normal-order evaluation





# Strict vs. lazy

- Strict evaluation:
  - is simple and easy to understand
  - may do unnecessary computations
  - may not terminate on well-defined problems
- Lazy evaluation:
  - only does as much work as is needed
  - can give results where strict evaluation does not
  - can complicate reasoning about efficiency
    - Will this expression be evaluated? If so, when?
- Haskell uses lazy evaluation





# Haskell's evaluation strategy

- In Haskell, the evaluation strategy is:
  - only reduce an expression if necessary (*i.e.* if some other expression needs it for its evaluation)
  - reduce the *outermost* redex first
  - if a redex is duplicated, reducing it in one place reduces it in all other places (they are the same expression)





# Why lazy?

- *Why* does Haskell use lazy evaluation?
  - Almost every other programming language ever invented uses strict evaluation!





# Why lazy?

- Reason 1: Haskell is designed to make equational reasoning as natural as possible
- Equational reasoning is simpler with a lazy evaluation model
  - Like high school algebra (substitute equals for equals, simplify)
  - Strict evaluation sometimes requires you to jump through hoops to get the effect you want (e.g. Y combinator in CS 4 lecture)





# Why lazy?

- Reason 2: Lazy evaluation has better modularity properties than strict evaluation
- Many functions are composed from other functions
- Some functions may generate large data structures, pass them to other functions, which then filter out parts they don't need
- This is much more natural/efficient in a lazy language (as we'll see)
- Reference: Hughes, Why Functional Programming Matters





# Why lazy?

- Reason 3: From Simon Peyton-Jones, lead developer of GHC Haskell compiler:  
*"Lazy evaluation keeps you honest!"*





# Why lazy?

- Meaning: Haskell is intended to be a *pure* functional language
  - i.e. no (uncontrolled) side effects
- Lazy evaluation means that evaluation order of function arguments is not known in advance
- This would be extremely problematic in the presence of side effects!





# Why lazy?

- With lazy evaluation:
  - you can't depend on the evaluation order to be predictable
  - you can't have arguments to functions being side-effecting if you intend the side effects to occur in a particular order (which you almost always do)
  - so you are forced to retain purity!
  - (And find a different way to deal with side effects.)





# Downside to laziness

- Laziness *simplifies* equational reasoning, but it *complicates* reasoning about time and (especially) space efficiency of functions
- We will see examples of this as we proceed
- Haskell also has ways of controlling lazy evaluation on an argument-by-argument basis, which we'll see later too





## Example 2

```
infinity :: Integer  
infinity = infinity + 1
```

- Try to evaluate:

```
infinity
```

```
→ infinity + 1
```

```
→ (infinity + 1) + 1
```

```
→ ((infinity + 1) + 1) + 1
```

- Evaluation never terminates!
- The expression infinity has no normal form!
- Non-terminating expressions called *bottom* (⊥)





## Example 3

```
three :: Integer -> Integer
```

```
three n = 3
```

- Try to evaluate **three infinity**:

```
three infinity
```

→ 3

- Evaluation is trivial using lazy evaluation strategy
- Cannot evaluate using strict evaluation strategy!
- Guarantee: if both lazy and strict evaluations terminate, they give the same result





## Example 3

- Recall that non-terminating expressions like infinity are denoted by ⊥ (*bottom*)
- Definition of lazy/strict functions:
  - if **f** ⊥ == ⊥, the function is strict
  - otherwise (like **three**) the function is lazy
- Strict functions require that their arguments be evaluated before proceeding
- Even Haskell has some strict functions
  - e.g. built-in arithmetic operations (+ - \* / on **Ints/Integers** etc.)





# Example 4

- Good old factorial:

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

- We will evaluate **factorial 3**
- We'll assume that something needs this result, otherwise it will just stay as (unevaluated)  
**factorial 3**





# Example 4

- **factorial 3**
  - doesn't match **factorial 0**, continue...
  - matches **factorial n** with **n == 3**
  - evaluate **n \* factorial (n - 1)** with **n == 3**
  - evaluate **3 \* factorial (3 - 1)**
  - **\*** on **Integers** is strict in both arguments (built-in operator)
  - need to evaluate **factorial (3 - 1)**
  - pattern matching here *requires* that we evaluate **(3 - 1)** so we can tell if this matches **0** or not
  - evaluate **3 - 1 → 2**
  - Continued...





# Example 4

- Continuing...
  - evaluate `factorial 2` → `2 * factorial (2 - 1)`
  - Note: full expression now is:
    - `3 * (2 * factorial (2 - 1))`
  - evaluate `2 * factorial (2 - 1)`
  - → `2 * factorial 1`
  - → `2 * (1 * factorial (1 - 1))`
  - → `2 * (1 * factorial 0)`
  - Recall: `factorial 0` reduces to `1`
  - → `2 * (1 * 1)`
  - Continuing...





# Example 4

- Continuing...
  - Recall pending operation:
    - `3 * (2 * factorial (2 - 1))`
    - $\rightarrow 3 * (2 * (1 * 1))$
    - $\rightarrow 3 * (2 * 1)$
    - $\rightarrow 3 * 2$
    - $\rightarrow 6$





# Example 4

- Notes on this example:
  - Most of it was very simple
  - Just high school algebra: substitute equals for equals, simplify
  - Tricky parts:
    - Knowing which operators/functions are strict
      - e.g. `*` is strict in its arguments
    - Knowing when evaluation must be forced
- Lazy evaluation is one of the conceptually hardest features of Haskell!
  - but can also be very useful!





# Example 5

- Recall tail-recursive factorial function:

```
factorial_tr :: Integer -> Integer
```

```
factorial_tr n = iter n 1
```

where

```
iter :: Integer -> Integer -> Integer
```

```
iter 0 r = r
```

```
iter n r = iter (n - 1) (n * r)
```

- Let's evaluate `factorial_tr 3`





# Example 5

```
factorial_tr 3
```

→ **iter** 3 1

- doesn't match **iter** 0 **r**, continue...
- matches **iter n r** with **n == 3**, substitute
  - **iter** (3 - 1) (3 \* 1)
- must reduce **3 - 1** to check pattern matching with **0**
  - **iter** 2 (3 \* 1)
- doesn't match **iter** 0 **r**, continue...
- matches **iter n r** with **n == 2, r = (3 \* 1)**, substitute
  - **iter** (2 - 1) (2 \* (3 \* 1))





# Example 5

- `iter (2 - 1) (2 * (3 * 1))`
- must evaluate `(2 - 1)` for pattern matching
  - `iter 1 (2 * (3 * 1))`
  - `iter (1 - 1) (1 * (2 * (3 * 1)))`
- must evaluate `(1 - 1)` for pattern matching
  - → `iter 0 (1 * (2 * (3 * 1)))`
  - → `(1 * (2 * (3 * 1)))`
  - → `(1 * (2 * 3))`
  - → `(1 * 6)`
  - → `6`





# Example 5

- Note that `iter` is strict in its first argument only
- Second argument not evaluated until `iter` is done and a value result is needed
- In fact, no computation at all is done unless the result is needed!
  - So `factorial_tr 3` won't be evaluated unless you need to do something with the result (e.g. print it)
- Probably not the way you're used to thinking about how computations unfold





# Example 6

- Note that tail recursion *doesn't* make `factorial_tr` more space-efficient!
- This is a very sharp contrast to strictly-evaluated languages like OCaml
- There are ways to force strict evaluation if you want/need to, which we will see later in the course
- *Don't assume your OCaml knowledge can carry through unchanged when programming in Haskell!*





# Next time

- More Haskell basics
- Lists
- Polymorphic types
- Function composition
- Point-free and point-wise style

