

CompSci 790/657 – Spring 2020

- ▶ Instructor: Tian Zhao
- ▶ Office: EMS 1145
- ▶ Office Hour: TR 11AM – Noon.
- ▶ Email: tzhao@uwm.edu
- ▶ Homepage: <https://uwm.edu/canvas/home/>
- ▶ Time/Location: TR 5:30 PM – 6:45PM, EMS E220
 - ▶ Lecture notes
 - ▶ Homework
 - ▶ Announcements
- ▶ Grading policy:
 - ▶ Homework: 60% (late submission with 20% penalty per day)
 - ▶ Midterm: 20% (5:30 – 6:45 PM, Thursday, 03/12/2020, EMS E220)
 - ▶ Final: 20% (Take home)

Schedule

- ▶ Why functional programming?
- ▶ Haskell primitives: constant, variable, tuple, list, and function
- ▶ Lazy evaluation
- ▶ Types and type class
- ▶ Higher order functions
- ▶ Pattern matching and algebraic data type
- ▶ Functor, Applicative, Monoid, and Monad
- ▶ Parser derivation
- ▶ Monad transformer
- ▶ Program verification
- ▶ Embedded DSL
- ▶ Type-based modeling
- ▶ Software transactional memory

Why study programming languages?

- ▶ Understand language designs to use it effectively
 - ▶ Functional vs. Imperative
 - ▶ Procedural vs. Object-oriented
 - ▶ Compiled vs. Interpreted
- ▶ Learn how to learn a language quickly
 - ▶ Syntax (how to write a parsable program)
 - ▶ Semantics (how to write a program that runs correctly)
 - ▶ Idioms (how to write a program that runs efficiently)

Imperative languages

Java

```
1  int factorial (int n)  {  
2      int ret = 1;  
3      while(n > 0) {  
4          ret = ret * n;  
5          n = n - 1;  
6      }  
7      return ret;  
8  }
```

What makes it imperative?

- ▶ Assignments (updates)
- ▶ Iterations (loops)
- ▶ Order of execution is important

Functional languages

Haskell

```
1 factorial 0 = 1
2 factorial n = n * factorial (n-1)
```

What makes it functional?

- ▶ Single-valued variables (no assignments)
- ▶ Recursions (no iterations)
- ▶ Order of execution does not matter

Functional languages

SML

```
1  fun factorial 0 = 1
2  | factorial n = n * factorial (n-1)
```

What makes it functional?

- ▶ Single-valued variables (no assignments)
- ▶ Recursions (no iterations)
- ▶ Order of execution does not matter

How to select a programming language?

- ▶ A broad overview of the language
 - ▶ Language style?
 - ▶ Applicable domain?
 - ▶ Level of support?
- ▶ Suitability to task at hand
 - ▶ Productivity (domain specificity, library support, tooling)
 - ▶ Maintainability (system compatibility, scalability, modularity, portability)
 - ▶ Efficiency (language runtime efficiency)
- ▶ Connection to programming language concepts
 - ▶ Level of abstraction: function, object (class), string, higher-order function, interface, trait (mixin)
 - ▶ Statically vs dynamically typed. Strong vs weak typing
 - ▶ Compiled to binary or bytecode
 - ▶ Functional vs imperative (with or without assignments, recursion vs. iteration)
 - ▶ Automatic vs manual memory management

How to learn a programming language?

- ▶ **Syntax:** how to write a parsable program: BNF grammar, lexical and phrase structure, regular expression, parse tree, abstract syntax tree, abstract syntax.
- ▶ **Semantics:** how to write a correct program.
Static semantics (type system), dynamic semantics (operational vs denotational semantics).
- ▶ **Idioms:** how to write an efficient program.
Recursion vs iteration; Value vs. variable; Functions vs. objects; Pattern matching vs. branching; Inheritance vs. composition; Type inference vs type declaration; higher-order vs. first-order function;
- ▶ **Tools:** compiler, interpreter, libraries, and debugger

Why functional programming?

- ▶ In functional programs, there is
 - ▶ no assignment
 - ▶ no side effects
 - ▶ no control flow
- ▶ In functional programs,
 - ▶ the order of execution is irrelevant
 - ▶ there is referential transparency (free to exchange variables with their values)
 - ▶ meaning is more tractable mathematically

Why functional programming?

- ▶ Functional programmers are more productive
- ▶ Functional programs are an order of magnitude smaller than conventional programs.
- ▶ Key advantage is modularity – modular design means higher productivity (program in the large)
- ▶ Evolution of programming:
Assembly \Rightarrow Structured programming (no GoTo) \Rightarrow
Functional programming (no assignment)

Why functional programming?

Modularity through function composition.

Example in JavaScript (in functional style):

```
1 let lst = [1,2,3,4]
2
3 let sum = x => x.reduce((a,b)=>a+b, 0)
4
5 sum(lst) // 10
6
7 let prod = x => x.reduce((a,b)=>a*b, 1)
8
9 prod(lst) // 24
```

Why functional programming?

Modularity through function composition.

Example in JavaScript (in functional style):

```
1 let lst = [1,2,3,4]
2
3 let even = x => x%2 == 0
4
5 let forall = f => x => x.reduce((c, e) => c && f(e), true)
6
7 forall (even) (lst) // false
8
9 let exists = f => x => x.reduce((c, e) => c || f(e), false)
10
11 exists (even) (lst) // true
```

Why functional programming?

Modularity through function composition.
Example in Haskell

```
1 lst = [1, 2, 3, 4]
2
3 sum = foldl (+) 0
4
5 sum lst -- 10
6
7 prod = foldl (*) 1
8
9 prod lst -- 24
```

Why functional programming?

Modularity through function composition.
Example in Haskell

```
1 lst = [1, 0.2, 3, 4]
2
3 sum = foldl (+) 0
4
5 sum lst -- 8.2
6
7 prod = foldl (*) 1
8
9 prod lst -- 2.4
```

Why functional programming?

Modularity through function composition.

Example in Haskell

```
1 lst = [1, 2, 3, 4]
2
3 even = \x -> mod x 2 == 0
4
5 forall f = foldl (\c e -> c && f e) True
6
7 forall even lst -- False
8
9 exists f = foldl (\c e -> c || f e) False
10
11 exists even lst -- True
```

Why functional programming?


Referential transparency

...the meaning of an expression is its value and there are no other effects, hidden or otherwise, in any procedure for actually obtaining it. Furthermore, the value of an expression depends only on the values of its constituent expressions...

Why functional programming?

Referential transparency

An expression may contain certain 'names' which stand for unknown quantities, but it is normal in mathematical notation to presume that different occurrences of the same name refer to the same unknown quantity...¹

¹Introduction to Functional Programming by R. Bird and P. Walder (page 4) 

Why functional programming?

Equational reasoning – expressions can be freely changed by others that denote the same value.

```
1 reverse (reverse lst) = lst
```

Two expressions are equivalent if and only if

- ▶ the both evaluate to the same value, or
- ▶ they both loop forever, or
- ▶ they both raise the same exception.

Why functional programming?

Equational reasoning – expressions can be freely changed by others that denote the same value.

```
1 replicate 4 True -- return [True, True, True, True]
```

We know 'replicate' is correct because:

```
1 -- `repeat x` returns [x, x, ... ]  
2 -- `take n l` returns the first n elements of l  
3  
4 replicate n x = take n (repeat x)
```

Why Haskell?

- ▶ Function vs action – pure code is separated from computation which can affect the external world.
- ▶ Lazy evaluation
- ▶ Strongly typed and global type inference
- ▶ Type class

Why Haskell?

Function vs action – pure code is separated from computation which can affect the external world.

- ▶ Real applications interact with external world
 - producing side effects.
- ▶ So how can Haskell program stay pure?
Answer: types

Why Haskell?

Function vs action – pure code is separated from computation which can affect the external world.

```
1 hello :: String -> IO ()
2 hello s = putStrLn ("Hi, " ++ s ++ "!")
3
4 hello "Tian" -- prints "Hi, Tian!"
```

The 'hello' function returns an IO action that, when executed, produces nothing and causes side effects (printing).

Why Haskell?

Function vs action – pure code is separated from computation which can affect the external world.

```
1 hi :: IO ()
2 hi = do putStrLn "Hello! What is your name?"
3         name <- getLine
4         putStrLn ("Hi, " ++ name ++ "!")
5
6 hi -- prints "Hello! What is your name?"
7    -- types   Tian
8    -- prints  "Hi, Tian!"
```

The 'hi' function takes no input and returns an IO action with side effects (reading and printing).

Why Haskell?

Actions are first class – can be manipulated like other values.

```
1 p :: IO ()
2 p = sequence_ [print 1, print True]
3
4 p
5 -- prints 1
6 --      True
```

`sequence_` takes a list of IO actions and return a composed IO action.

Why Haskell?

Actions are first class – can be manipulated like other values.

```
1 printTable :: [String] -> IO ()
2
3 printTable xs =
4     sequence_ [
5         putStrLn (show i ++ ":" ++ x)
6         | (x,i) <- xs `zip` [1..]
7     ]
8
9 printTable ["Why", "study", "Haskell?"]
10
11 -- prints 1:Why
12 --          2:study
13 --          3:Haskell?
```

where 'show' converts values to strings, 'zip' merges two lists, [1..] is an infinite list starting from 1, $(x, i) \leftarrow e$ selects each element (x, i) of 'e', and $[e \mid (x, i) \leftarrow e']$ is a list of 'e' for each (x, i) .

Why Haskell?

Lazy evaluation:

*"It makes it practical to modularise a program as a generator which constructs a large number of possible answers, and a selector which chooses the appropriate one."*²

Why Haskell?

Eager evaluation (SML)

```
1 (fn x y => if (x > 0) then y + 1 else 1) 0 (factorial 20)
2 -->
3 (fn x y => if (x > 0) then y + 1 else 1) 0 2432902008176640000
4 -->
5 if (0 > 0) then 2432902008176640000 + 1 else 1
6 -->
7 1
```

Why Haskell?

Lazy evaluation (Haskell)

```
1 (\x y -> if (x > 0) then y + 1 else 1) 0 (factorial 20)
2 -->
3 if (0 > 0) then (factorial 20) + 1 else 1
4 -->
5 1
```

Why Haskell?

```
1 xs `zip` [1..]
```

where 'zip' is the infix operator version of *zip*.

While [1..] is an infinite list, *zip* stops when *xs* ends.

Why Haskell?

```
1 fact_list x n = a : fact_list a (n+1)
2   where a = x * n
3
4 take 10 (drop 10 (fact_list 1 1))
5
6 -- prints [11!, 12!, ..., 20!]
7 -- [39916800,479001600,6227020800,87178291200,1307674368000,
8 --   20922789888000, 355687428096000,6402373705728000,
9 --   121645100408832000,2432902008176640000]
```

where `fact_list` evaluates to an infinite list of factorials

Why Haskell?

Type class is a systematic way of overloading, where type expression can resemble program expression.

```
1 class Size a where
2   (~~) :: a -> a -> Bool
3
4 data Length = Short | Medium | Long
```

Why Haskell?

```
1 class Size a where
2     (~~) :: a -> a -> Bool
3
4 data Length = Short | Medium | Long
5
6 instance Size Length where
7     Short ~~ Short = True
8     Short ~~ Medium = True
9     Medium ~~ Short = True
10    Medium ~~ Medium = True
11    Long ~~ Medium = True
12    Medium ~~ Long = True
13    Long ~~ Long = True
14    _ ~~ _ = False
```


Getting started

Install GHC and try ghci.

- ▶ ghci is Haskell interpreter (a console).
- ▶ ghc is Haskell compiler.
- ▶ ghci and ghc have slight differences at parsing and type inference.
- ▶ you can write a program in a text file and then run it through ghc.

Getting started

Get used to ghci.

- ▶ ghci is single-line by default (it runs each line you type right away).
- ▶ type `:set +m` to enable multi-line option
- ▶ type `:t e` to discover the type of `e`
- ▶ type `:i e` to display information of `e`.

Getting started

Rule of indentation:

Code which is part of some expression should be indented further in than the beginning of that expression

For example,

```
1 let
2   x = a
3   y = b
4
5 let i = a
6     j = b
7
8 case x of
9   p -> u
10  q -> v
```