

Roadmap

- > Functional vs. Imperative Programming
- > Pattern Matching
- > Referential Transparency
- > Lazy Evaluation
- > Recursion
- > Higher Order and Curried Functions



References

- > *“Conception, Evolution, and Application of Functional Programming Languages,”* Paul Hudak, ACM Computing Surveys 21/3, 1989, pp 359-411.
- > *“A Gentle Introduction to Haskell,”* Paul Hudak and Joseph H. Fasel
— www.haskell.org/tutorial/
- > *Haskell 2010 Language Report*
— www.haskell.org
- > *Real World Haskell*, Bryan O'Sullivan, Don Stewart, and John Goerzen
— book.realworldhaskell.org/read/

Conception, Evolution, and Application of Functional Programming Languages

<http://scgresources.unibe.ch/Literature/PL/Huda89a-p359-hudak.pdf>

A Gentle Introduction to Haskell

<https://www.haskell.org/tutorial/>

Haskell 2010 Language Report

<https://www.haskell.org/onlinereport/haskell2010/>

Real World Haskell

<http://book.realworldhaskell.org>

Roadmap



- > **Functional vs. Imperative Programming**
- > Pattern Matching
- > Referential Transparency
- > Lazy Evaluation
- > Recursion
- > Higher Order and Curried Functions

A Bit of History

<i>Lambda Calculus</i> (Church, 1932-33)	formal model of computation
<i>Lisp</i> (McCarthy, 1960)	symbolic computations with lists
<i>APL</i> (Iverson, 1962)	algebraic programming with arrays
<i>ISWIM</i> (Landin, 1966)	let and where clauses; equational reasoning; birth of “pure” functional programming ...
<i>ML</i> (Edinburgh, 1979)	originally meta language for theorem proving
<i>SASL, KRC, Miranda</i> (Turner, 1976-85)	lazy evaluation
<i>Haskell</i> (Hudak, Wadler, et al., 1988)	“Grand Unification” of functional languages ...

Church's **lambda calculus** predates computers, but is an influential early model of computation that has deeply influenced programming language design.

Lisp is a language that unifies data and programs. Functions are represented as lists that resemble lambda expressions and are then interpreted or compiled.

APL is a language for manipulating arrays and arrays of arrays. Programs are built up of functional operators applied to arrays. Later languages like Mathematica owe a great deal to APL.

ISWIM is a paper language design that has strongly influenced the design of functional languages like Haskell.

ML was designed as a theorem-proving meta-language, but ended up being a general purpose functional language.

Miranda and friends introduced “lazy evaluation” to the functional paradigm, though the idea dates from lambda calculus.

Haskell unified and cleaned up many of these ideas.

Programming without State

Imperative style:

```
n := x;  
a := 1;  
while n>0 do  
begin a:= a*n;  
      n := n-1;  
end;
```

Declarative (functional) style:

```
fac n =  
  if    n == 0  
  then  1  
  else  n * fac (n-1)
```

*Programs in pure functional languages have no explicit state.
Programs are constructed entirely by composing expressions.*

Note that the functional style resembles much more the typical mathematical (i.e., declarative) definition.

Pure Functional Programming Languages

Imperative Programming:

> Program = Algorithms + Data

Functional Programming:

> Program = Functions \circ Functions

What is a Program?

— A program (computation) is a *transformation* from input data to output data.

Key features of pure functional languages

1. All programs and procedures are *functions*
2. There are *no variables or assignments* — only input parameters
3. There are *no loops* — only recursive functions
4. The value returned by a function *depends only on the values of its parameters*
5. Functions are *first-class values*

Note that early functional languages like Lisp, APL and ML are not “pure”, that is they allow programs to have a modifiable state. Similarly Scala, a fusion of functional and object-oriented programming, is necessarily impure, as it builds on Java. Nevertheless, it is possible to write pure (stateless) programs even in impure languages.

What is Haskell?

Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on lazy functional languages.

— The Haskell 98 report

The *highlighted phrases* are key:

Haskell is intended to be a *general-purpose*, i.e., practical, language for arbitrary application domains. At the same time it is *purely functional* (i.e., stateless), so it is an explicit challenge to demonstrate that such languages can be practical.

Higher-order functions treat functions as first-class values, so can take functions as arguments and can yield functions as return values.

Non-strict semantics (as we shall see) means that expressions are evaluated lazily, i.e., values are only computed as needed. This enables highly expressive language features such as infinite lists.

...

...

Static polymorphic typing means (i) that programs are *statically* type-checked, i.e., before running them, and (ii) functions may be *polymorphic*, i.e., can take arguments of different types. In addition, Haskell supports (ML-style) *type inference*: (most) programs can be type-checked even without explicit type annotations.

User-defined algebraic datatypes are abstract data types that bundle functions together with a hidden representation (like OO classes, but without inheritance).

Pattern-matching offers a highly expressive way to define multiple cases for function definition (similar to Prolog).

Finally, *list comprehensions* offer a convenient mathematical set-like notation for defining lists of values.

“Hello World” in Haskell

```
hello() = print "Hello World"
```

`hello` is a function that takes an empty tuple as an argument. It invokes the `print` function with the string (character array) `"hello world"` as its argument.

You may well ask, “If Haskell is ‘pure’, then how can you print something without having a side effect?”

Well, consider “output” as an infinite lazy list that is being computed over time ...

To run Haskell programs, download and install the Glasgow Haskell Platform:

<https://www.haskell.org/platform/>

To define and run code interactively, start the ghci interpreter. You can define functions in a file and *load* them, or define them interactively with `let`:

```
% ghci
```

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
```

```
Prelude> let hello () = print "hello"
```

```
Prelude> hello ()
```

```
"hello"
```

```
Prelude> :load Intro.hs
```

```
[1 of 2] Compiling HUnit           ( HUnit.hs, interpreted )
```

```
[2 of 2] Compiling Main             ( Intro.hs, interpreted )
```

```
Ok, modules loaded: HUnit, Main.
```

```
*Main> fac 5
```

```
120
```

You can also change the prompt with the command

```
:set prompt "% "
```

Roadmap

- > Functional vs. Imperative Programming
- > **Pattern Matching**
- > Referential Transparency
- > Lazy Evaluation
- > Recursion
- > Higher Order and Curried Functions



Pattern Matching

Haskell supports multiple styles for specifying case-based function definitions:

Patterns:

```
fac' 0 = 1
fac' n = n * fac' (n-1)

-- or: fac' (n+1) = (n+1) * fac' n
```

Guards:

```
fac'' n | n == 0 = 1
        | n >= 1 = n * fac'' (n-1)
```

The evaluation order of unguarded patterns can be significant.
Note that either constants or variables can appear in patterns.

What happens if you try to evaluate this?:

```
fac' (-1)
```

Lists

Lists are *pairs* of elements and lists of elements:

- > `[]` — stands for the empty list
- > `x:xs` — stands for the list with `x` as the *head* and `xs` as the *tail* (rest of the list)

The following short forms make lists more convenient to use

- > `[1, 2, 3]` — is syntactic sugar for `1:2:3:[]`
- > `[1..n]` — stands for `[1, 2, 3, ... n]`

Lists in Haskell are *homogeneous*, that is they can only contain elements of a single type. We will discuss type in more detail in the next lecture.

Using Lists

Lists can be *deconstructed* using patterns:

```
head (x:_) = x

len [] = 0
len (_:xs) = 1 + len xs

prod [] = 1
prod (x:xs) = x * prod xs

fac ' ' n = prod [1..n]
```

The underscore (`_`) is a wildcard pattern and matches anything. In the definition of `head`, it says, “I am interested in the value of the head and call it `x`; I don’t care what the tail is.”

Note that `head [1..5] = head (1:2:3:4:5) = 1`

What is `head []` ?

Note that `len` is defined recursively. Pure functional languages tend to use recursion rather than explicit loops (though loops can be defined in Haskell as a utility function).

List comprehensions

A list comprehension uses a set-like notation to define a list:

```
[ x*x | x <- [1..10] ]
```

```
⇒ [1,4,9,16,25,36,49,64,81,100]
```

List comprehensions follow the general form:

```
[ elements | definition ]
```

Where *elements* are Haskell expressions (e.g., tuples, lists etc) containing variables defined to the right.

See:

https://wiki.haskell.org/List_comprehension

Roadmap

- > Functional vs. Imperative Programming
- > Pattern Matching
- > **Referential Transparency**
- > Lazy Evaluation
- > Recursion
- > Higher Order and Curried Functions



Referential Transparency

A function has the property of referential transparency if *its value depends only on the values of its parameters*.

 Does $f(x) + f(x)$ equal $2 * f(x)$? In C? In Haskell?

Referential transparency means that “*equals can be replaced by equals*”.

In a pure functional language, all functions are referentially transparent, and therefore *always yield the same result* no matter how often they are called.

Evaluation of Expressions

Expressions can be (formally) evaluated by substituting arguments for formal parameters in function bodies:

```
fac 4
⇒ if 4 == 0 then 1 else 4 * fac (4-1)
⇒ 4 * fac (4-1)
⇒ 4 * (if (4-1) == 0 then 1 else (4-1) * fac (4-1-1))
⇒ 4 * (if 3 == 0 then 1 else (4-1) * fac (4-1-1))
⇒ 4 * ((4-1) * fac (4-1-1))
⇒ 4 * ((4-1) * (if (4-1-1) == 0 then 1 else (4-1-1) * ...))
⇒ ...
⇒ 4 * ((4-1) * ((4-1-1) * ((4-1-1-1) * 1)))
⇒ ...
⇒ 24
```

Of course, real functional languages are not implemented by syntactic substitution ...

As we shall see in the lecture on lambda calculus, the semantics of function application can indeed be defined formally as syntactic substitution of arguments in the body of the function (while taking care to avoid name clashes).

Roadmap

- > Functional vs. Imperative Programming
- > Pattern Matching
- > Referential Transparency
- > **Lazy Evaluation**
- > Recursion
- > Higher Order and Curried Functions



Lazy Evaluation

“Lazy”, or “normal-order” evaluation only evaluates expressions when they are actually needed. Clever implementation techniques (Wadsworth, 1971) allow replicated expressions to be shared, and thus avoid needless recalculations.

So:

```
sqr n = n * n
```

```
sqr (2+5) ⇨ (2+5) * (2+5) ⇨ 7 * 7 ⇨ 49
```

Lazy evaluation allows some functions to be evaluated even if they are passed incorrect or non-terminating arguments:

```
ifTrue True x y = x  
ifTrue False x y = y
```

```
ifTrue True 1 (5/0) ⇨ 1
```

Again, as we shall see with the lambda calculus, functions can either be evaluated *strictly*, by evaluating all arguments first (i.e., especially if the arguments are complex expressions, not primitive values), or *lazily*, by evaluating arguments only if and when they are needed.

Conventional languages (like Java) are strict, while pure functional languages are lazy. Nevertheless, one can program in a lazy style in any language, even Java.

Strict evaluation is also known as *applicative evaluation*, and lazy evaluation is known as *normal order evaluation*, for reasons that will become clear later (only normal order is guaranteed to lead to a normalized value, if one exists).

Lazy Lists

Lazy lists are *infinite data structures* whose values are generated by need:

```
from n = n : from (n+1)
```

from 100 ⇨ *[100,101,102,103,...]*

```
take 0 _ = [ ]
```

```
take _ [ ] = [ ]
```

```
take (n+1) (x:xs) = x : take n xs
```

take 2 (from 100)

⇨ take 2 (100:from 101)

⇨ 100:(take 1 (from 101))

⇨ 100:(take 1 (101:from 102))

⇨ 100:101:(take 0 (from 102))

⇨ 100:101:[]

⇨ *[100,101]*

NB: The lazy list (from n) has the special syntax: [n..]

Lazy lists are a built-in feature of pure functional languages that derive directly from their lazy evaluation strategy.

One can easily simulate a lazy list in an OO language: define an object that knows how to compute and return its n th value, and caches all values computed thus far.

Programming lazy lists

Many sequences are naturally implemented as lazy lists.

Note the top-down, declarative style:

```
fibs = 1 : 1 : fibsFollowing 1 1
      where fibsFollowing a b =
              (a+b) : fibsFollowing b (a+b)
```

```
take 10 fibs
```

```
⇒ [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

 *How would you re-write fibs so that (a+b) only appears once?*

Note that `fibs` is an infinite list. We can freely manipulate it, as long as we don't try to compute all the values. The `take` function only forces the computation of a finite subsequence, and then “throws away” the rest.

Declarative Programming Style

```
primes = primesFrom 2
primesFrom n = p : primesFrom (p+1)
  where p = nextPrime n
nextPrime n
  | isPrime n           = n
  | otherwise          = nextPrime (n+1)
isPrime 2              = True
isPrime n              = notDivisible primes n
notDivisible (k:ps) n
  | (k*k) > n          = True
  | (mod n k) == 0     = False
  | otherwise          = notDivisible ps n
```

```
take 100 primes ⇨ [ 2, 3, 5, 7, 11, 13, ... 523, 541 ]
```

Roadmap

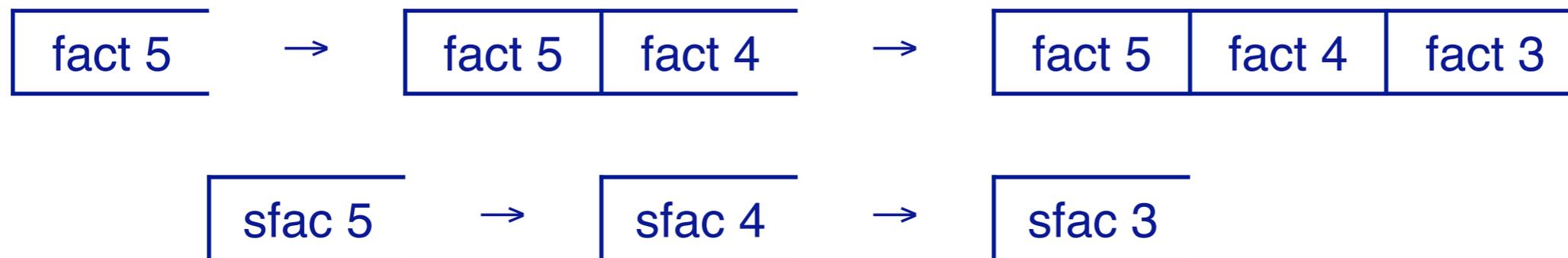
- > Functional vs. Imperative Programming
- > Pattern Matching
- > Referential Transparency
- > Lazy Evaluation
- > **Recursion**
- > Higher Order and Curried Functions



Tail Recursion

Recursive functions can be less efficient than loops because of the *high cost of procedure calls* on most hardware.

A tail recursive function calls itself only as its last operation, so the recursive call can be *optimized away* by a modern compiler since it needs only a single run-time stack frame:



Tail Recursion ...

A recursive function can be converted to a tail-recursive one by representing partial computations as explicit function parameters:

```
sfac s n = if    n == 0
           then  s
           else  sfac (s*n) (n-1)
```

```
sfac 1 4 ⇨ sfac (1*4) (4-1)
           ⇨ sfac (1*4) 3
           ⇨ sfac (1*4*3) (3-1)
           ⇨ ...
           ⇨ sfac (1*4*3*2*1) 0
           ⇨ (1*4*3*2*1)
           ⇨ ...
           ⇨ 24
```

Recall that the last step of `fac n` was `n * fac (n-1)`. In order to transform `fac` into a tail-recursive function, we must turn the rest of the computation (`n * ...`) into a parameter. This is exactly what we do by adding the parameter `s` to the function `sfac`: it accumulates the progressive multiplications.

In general this is what you need to do make recursive functions tail-recursive.

Note that the value of `s` is not needed until the end, so it is computed lazily.

Multiple Recursion

Naive recursion may result in unnecessary recalculations:

```
fib 1      = 1
fib 2      = 1
fib (n+2)  = fib n + fib (n+1)  – NB: Not tail-recursive!
```

Efficiency can be regained by *explicitly passing calculated values:*

```
fib' 1      = 1
fib' n      = a
              where (_,a) = fibPair n
fibPair 1    = (0,1)
fibPair n    = (b,a+b)
              where (a,b) = fibPair (n-1)
```

 *How would you write a tail-recursive Fibonacci function?*

Note that `fibPair` expresses the `n`th Fibonacci pair. By encapsulating a pair of values, everything is available to compute the next pair, so only one recursive step is needed.

Roadmap



- > Functional vs. Imperative Programming
- > Pattern Matching
- > Referential Transparency
- > Lazy Evaluation
- > Recursion
- > **Higher Order and Curried Functions**

Higher Order Functions

Higher-order functions treat other *functions as first-class values* that can be composed to produce new functions.

```
map f [ ]      = [ ]  
map f (x:xs)  = f x : map f xs
```

```
map fac [1..5]  
⇒ [1, 2, 6, 24, 120]
```

NB: `map fac` is a new function that can be applied to lists:

```
mfac l = map fac l
```

```
mfac [1..3]  
⇒ [1, 2, 6]
```

Note that we can also write simply:

```
mfac = map fac
```

As we shall see below, `mfac` and `map` are both *Curried* functions that take their arguments progressively.

Anonymous functions

Anonymous functions can be written as “lambda abstractions”.
The function $(\lambda x \rightarrow x * x)$ behaves exactly like `sqr`:

```
sqr x = x * x
```

```
sqr 10  
⇒ 100  
(\x -> x * x) 10  
⇒ 100
```

Anonymous functions are first-class values:

```
map (\x -> x * x) [1..10]  
⇒ [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Curried functions

A Curried function [named after the logician H.B. Curry] *takes its arguments one at a time*, allowing it to be treated as a higher-order function.

```
plus x y      = x + y      -- curried addition
```

```
plus 1 2  
⇒ 3
```

```
plus' (x,y)   = x + y      -- normal addition
```

```
plus' (1,2)  
⇒ 3
```

Understanding Curried functions

```
plus x y = x + y
```

is the same as:

```
plus x = \y -> x+y
```

In other words, plus is *a function of one argument that returns a function* as its result.

```
plus 5 6
```

is the same as:

```
(plus 5) 6
```

In other words, we invoke `(plus 5)`, obtaining a function,

```
\y -> 5 + y
```

which we then pass the argument 6, yielding 11.

Now we can see that `map` is a Curried function too.

`map f` *returns a function* that maps a list of elements to a list with `f` applied to those elements.

In particular:

```
map (\x -> x * x)
```

returns a function that maps a list of numbers to the list of squares of those numbers.

```
let sqrmap = map (\x -> x * x)
```

```
sqrmap [1..5]
```

```
[1,4,9,16,25]
```

Using Curried functions

Curried functions are useful because we can bind their arguments incrementally

```
inc = plus 1           -- bind first argument to 1
```

```
inc 2
```

```
⇒ 3
```

```
fac = sfac 1          -- binds first argument of
```

```
where sfac s n       -- a curried factorial
```

```
    | n == 0         = s
```

```
    | n >= 1        = sfac (s*n) (n-1)
```

Currying

The following (pre-defined) function takes a binary function as an argument and turns it into a curried function:

```
curry f a b = f (a, b)

plus(x,y)   = x + y           -- not curried!
inc         = (curry plus) 1

sfac(s, n)  = if n == 0       -- not curried
              then s
              else sfac (s*n, n-1)

fac = (curry sfac) 1         -- bind first argument
```

To be continued ...

- > Enumerations
- > User data types
- > Type inference
- > Type classes

What you should know!

-  *What is referential transparency? Why is it important?*
-  *When is a function tail recursive? Why is this useful?*
-  *What is a higher-order function? An anonymous function?*
-  *What are curried functions? Why are they useful?*
-  *How can you avoid recalculating values in a multiply recursive function?*
-  *What is lazy evaluation?*
-  *What are lazy lists?*

Can you answer these questions?

-  Why don't pure functional languages provide loop constructs?*
-  When would you use patterns rather than guards to specify functions?*
-  Can you build a list that contains both numbers and functions?*
-  How would you simplify fibs so that $(a+b)$ is only called once?*
-  What kinds of applications are well-suited to functional programming?*



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>