4. Functional Programming

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/
- *Report on the Programming Language Haskell 98 A Non-strict, Purely Functional Language*, Simon Peyton Jones and John Hughes [editors], February 1999
  - www.haskell.org
- *Real World Haskell*, Bryan O'Sullivan, Don Stewart, and John Goerzen
  - book.realworldhaskell.org/read/

Roadmap

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

A Bit of History

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

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.

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*

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

“Hello World” in Hugs

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

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)
```

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 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]`

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]
```

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 <u>referential transparency</u> 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 ...

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
```

Lazy Lists

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

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

take 0 _              = [ ]
take _ [ ]            = [ ]
take (n+1) (x:xs)     = x : take n xs
```

`from 100` ⇙ *`[100,101,102,103,....`*

`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..]`

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?

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]*

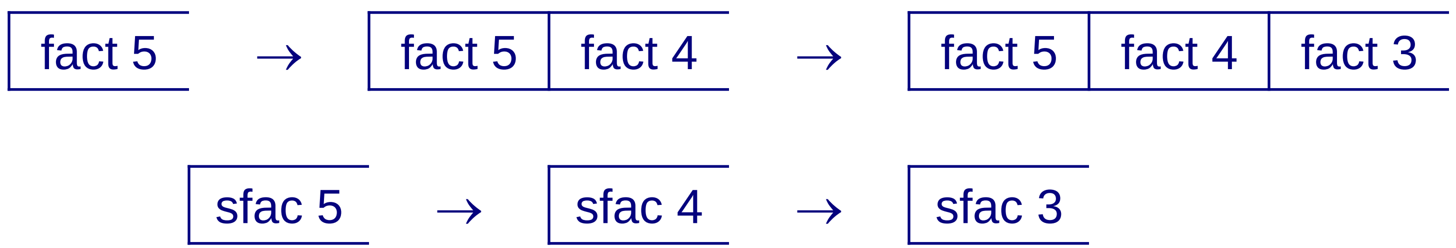
Roadmap

- 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 4 3
         ⇙  sfac (4*3) (3-1)
         ⇙  sfac 12 2
         ⇙  sfac (12*2) (2-1)
         ⇙  sfac 24 1
         ⇙  ...
         ⇙  24
```

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   = (1,0)
fibPair (n+2) = (a+b,a)
        where (a,b) = fibPair (n+1)
```

How would you write a tail-recursive Fibonacci function?

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 = map fac
```

```
mfac [1..3]
➘  [1, 2, 6]
```

Anonymous functions

Anonymous functions can be written as “lambda abstractions”.

The function `(\x -> 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`.

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?*

License