An Introduction to Programming in Haskell

Mark P Jones

Portland State University

Haskell Resources:

Haskell Resources:

- The focal point for information about Haskell programming, implementations, libraries, etc… is www.haskell.org
- I'll be using:
  - the Hugs interpreter (haskell.org/hugs)
  - the Glasgow Haskell compiler, GHC, and interpreter, GHCi (haskell.org/ghc)
- Online tutorials/references:
  - learnyouahaskell.com
  - book.realworldhaskell.org

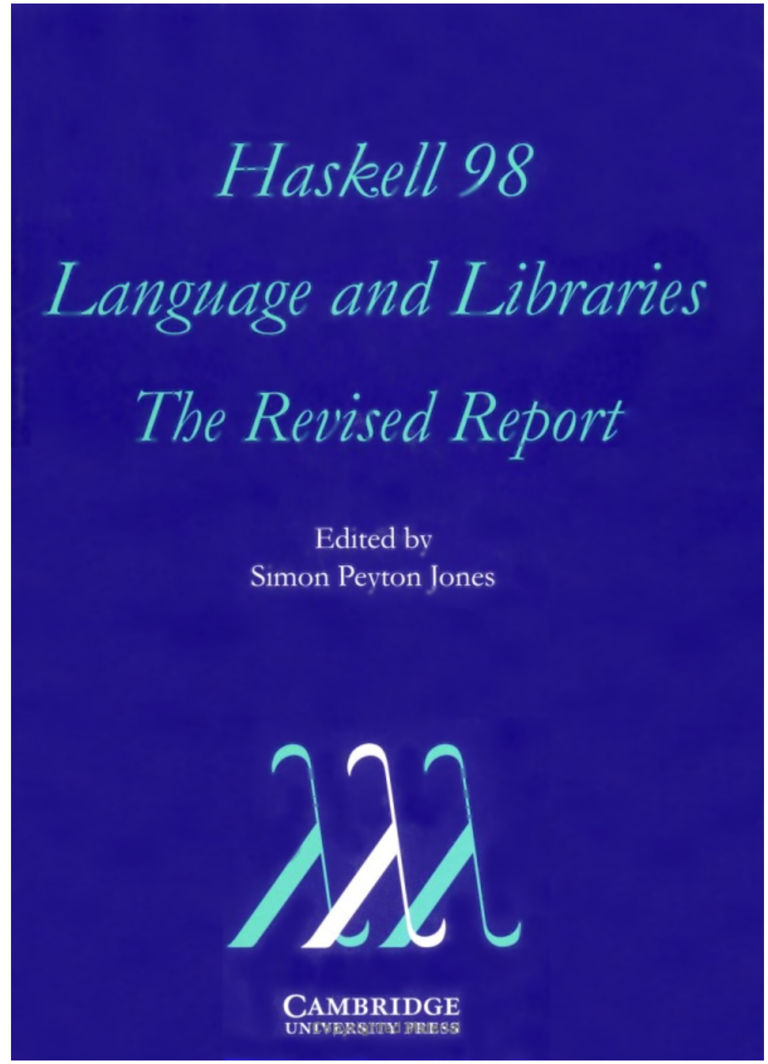
The Language Report:

The definition of the Haskell 98 standard

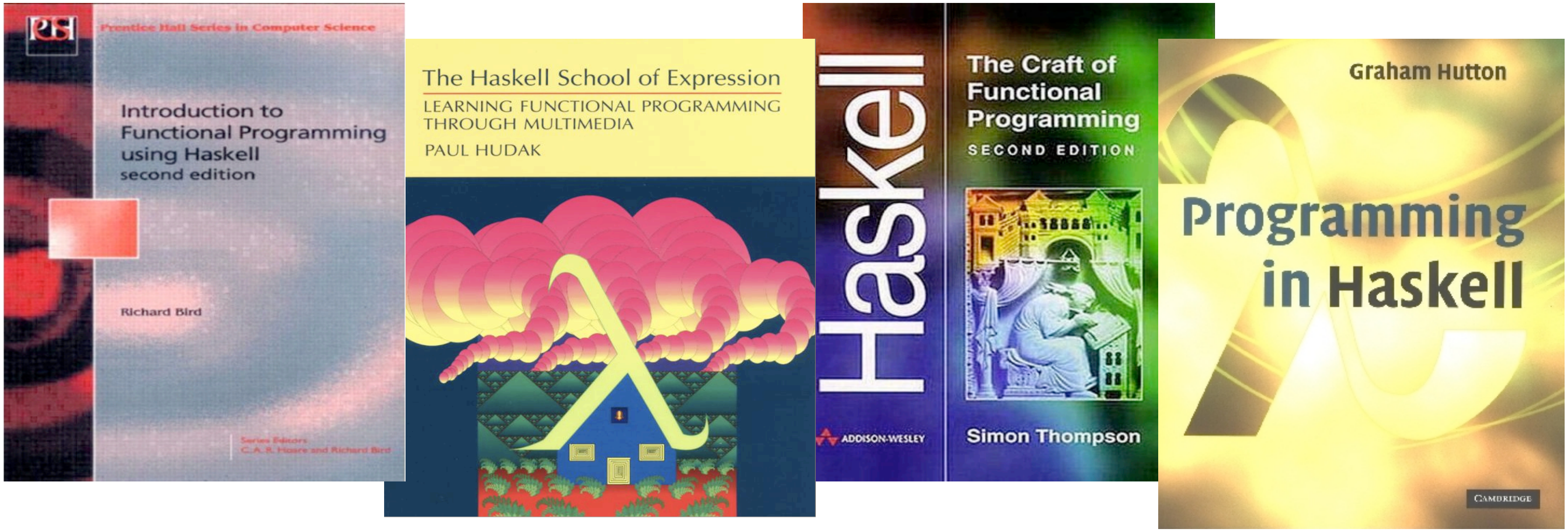
Lots of technical details … not a great read!

Available in hard copy from Cambridge University Press

Or in pdf/html/etc… from www.haskell.org/definition



Textbooks:



- *Introduction to Functional Programming using Haskell* (2nd edition), Richard Bird
- *The Haskell School of Expression*, Paul Hudak
- *Haskell: The Craft of Functional Programming* (2nd edition), Simon Thompson
- *Programming in Haskell*, Graham Hutton

What is Functional Programming?

What is Functional Programming?

- Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands

- Expressions are formed by using functions to combine basic values

- A functional language is a language that supports and encourages programming in a functional style

Functions:

In a pure functional language:

- The result of a function depends *only* on the values of its inputs:
  - Like functions in mathematics
  - No global variables / side-effects
- Functions are first-class values:
  - They can be stored in data structures
  - They can be passed as arguments or returned as results of other functions

Example:

- Write a program to add up the numbers from 1 to 10

In C, C++, Java, C#, ... :

initialization

initialization

iteration

```
int tot = 0;
for (int i=1; i<10; i++)
  tot = tot + i;
```

update

update

implicit result returned in the variable `tot`

In ML:

accumulating parameter

```
let fun sum i tot
      = if i>10
          then tot
          else sum (i+1) (tot+i)
in sum 1 0
end
```

(tail) recursion

initialization

result is the value of this expression

In Haskell:

```
sum [1..10]
```

combining function

the list of numbers to add

result is the value of this expression

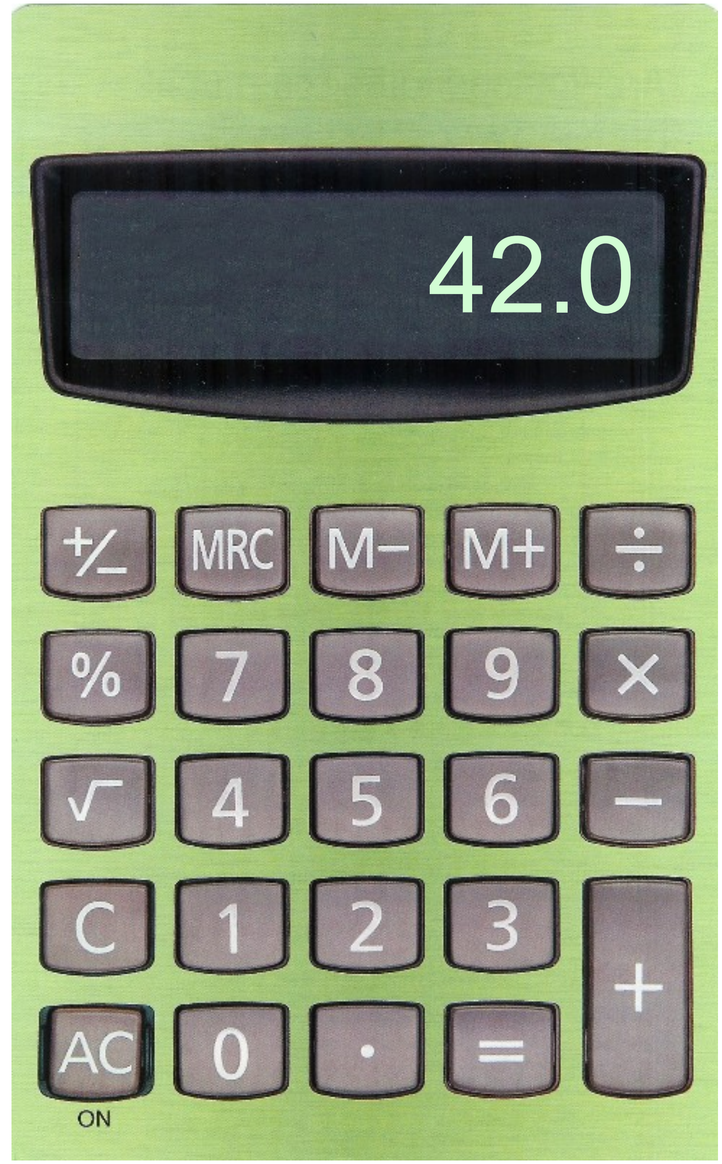
Raising the Level of Abstraction:

"If you want to reduce [design time], you have to stop thinking about something you used to have to think about." (Joe Stoy, recently quoted on the Haskell mailing list)

- Example: memory allocation
- Example: data representation
- Example: order of evaluation
- Example: (restrictive) type annotations

Computing by Calculating:

- Calculators are a great tool for manipulating numbers
- Buttons for:
  - entering digits
  - combining values
  - using stored values
- Not so good for manipulating large quantities of data
- Not good for manipulating other types of data



Computing by Calculating:

- What if we could "calculate" with other types of value?

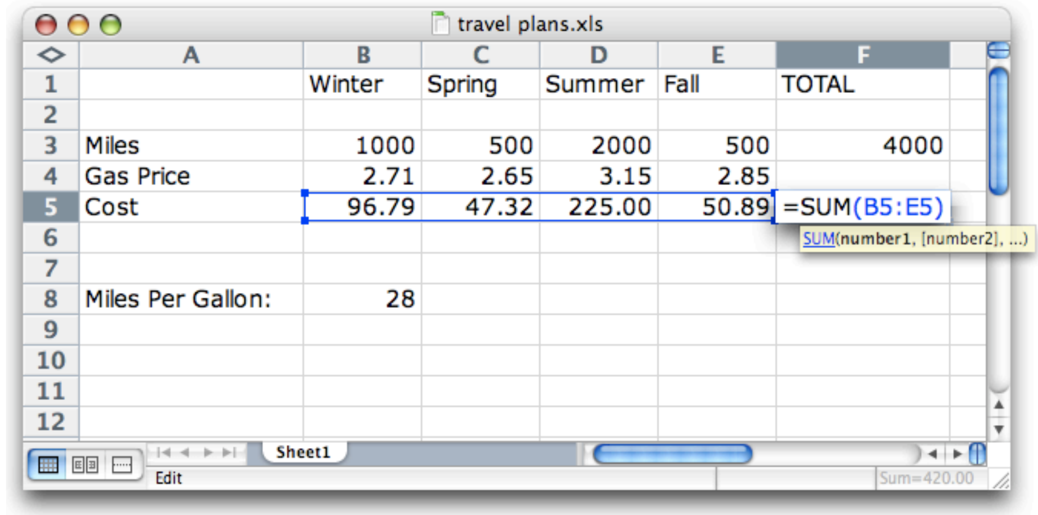
- Buttons for:
  - entering pixels
  - combining pictures
  - using stored pictures

- I wouldn't want to calculate a whole picture this way!

- I probably want to deal with *several different types* of data *at the same time*

Computing by Calculating:

- Spreadsheets are better suited for dealing with larger quantities of data



- Values can be named (but not operations)
- Calculations (i.e., programs) are recorded so that they can be repeated, inspected, modified
- Good if data fits an "array"
- Not so good for multiple types of data

Functional Languages:

- Multiple types of data
  - Primitive types, lists, functions, …
  - Flexible user defined types …
- Operations for combining values to build new values (combinators)
- Ability to name values and operations (abstraction)
- Scale to arbitrary size and shape data
- "Algebra of programming" supports reasoning

Getting Started with Haskell

Starting Hugs:

```
user$ hugs

__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs>
```

The most important commands:

- :q quit
- :l file load file
- :e file edit file
- Expr evaluate expression

The read-eval-print loop:

1. Enter expression at the prompt
2. Hit return
3. *The expression is read, checked, and evaluated*
4. *Result is displayed*
5. Repeat at Step 1

Simple Expressions:

Expressions can be constructed using:

- The usual arithmetic operations:

  1 + 2 * 3

- Comparisons:

  1 == 2 'a' < 'z'

- Boolean operators:

  True && False not False

- Built-in primitives:

  odd 2 sin 0.5

- Parentheses:

  odd (2 + 1) (1 + 2) * 3

- Etc …

Expressions Have Types:

- The type of an expression tells you what kind of value you will get when you evaluate that expression:
- In Haskell, read "::" as "has type"
- Examples:
  - 1 :: Int, 'a' :: Char, True :: Bool, 1.2 :: Float, …
- You can even ask Hugs for the type of an expression: :t expr

Type Errors:

```
Hugs> 'a' && True
ERROR - Type error in application
*** Expression      : 'a' && True
*** Term            : 'a'
*** Type            : Char
*** Does not match  : Bool

Hugs> odd 1 + 2
ERROR - Cannot infer instance
*** Instance    : Num Bool
*** Expression : odd 1 + 2

Hugs>
```

Pairs:

- A pair packages two values into one

  (1, 2)   ('a', 'z')   (True, False)

- Components can have different types

  (1, 'z')   ('a', False)   (True, 2)

- The type of a pair whose first component is of type A and second component is of type B is written (A,B)
- What are the types of the pairs above?

Operating on Pairs:

- There are built-in functions for extracting the first and second component of a pair:

```
fst (True, 2) = True
snd (0, 7)    = 7
```

Lists:

- Lists can be used to store zero or more elements, in sequence, in a single value:

  []   [1, 2, 3]   ['a', 'z']   [True, True, False]

- All of the elements in a list must have the same type
- The type of a list whose elements are of type A is written as [A]
- What are the types of the lists above?

Operating on Lists:

- There are built-in functions for extracting the head and the tail components of a list:
  - head [1,2,3,4] = 1
  - tail [1,2,3,4] = [2,3,4]

- Conversely, we can build a list from a given head and tail using the “cons” operator:
  - 1 : [2, 3, 4] = [1, 2, 3, 4]

More Operations on Lists:

- Finding the length of a list:
  length [1,2,3,4,5] = 5
- Finding the sum of a list:
  sum [1,2,3,4,5] = 15
- Finding the product of a list:
  product [1,2,3,4,5] = 120
- Applying a function to the elements of a list:
  map odd [1,2,3,4] = [True, False, True, False]

Continued …

- Selecting an element (by position):

  [1,2,3,4,5] !! 3 = 4

- Taking an initial prefix (by number):

  take 3 [1,2,3,4,5] = [1,2,3]

- Taking an initial prefix (by property):

  takeWhile odd [1,2,3,4,5] = [1]

- Checking for an empty list:

  null [1,2,3,4,5] = False

More ways to Construct Lists:

- Concatenation:

  [1,2,3] ++ [4,5] = [1,2,3,4,5]

- Arithmetic sequences:

  [1..10] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

  [1,3..10] = [1, 3, 5, 7, 9]

- Comprehensions:

  [2 * x | x <- [1,2,3,4,5]]   = [2, 4, 6, 8, 10]

  [y | y <- [1,2,3,4], odd y] = [1, 3]

Strings are Lists:

- A String is just a list of Characters

  ['w', 'o', 'w', '!'] = "wow!"

  ['a'..'j'] = "abcdefghij"

  "hello, world" !! 7 = 'w'

  length "abcdef" = 6

  "hello, " ++ "world" = "hello, world"

  take 3 "functional" = "fun"

Functions:

- The type of a function that maps values of type A to values of type B is written A -> B

- Examples:
  - odd :: Int -> Bool
  - fst :: (a, b) -> a   (a,b are type variables)
  - length :: [a] -> Int

Operations on Functions:

- Function Application. If f :: A -> B and x :: A, then f x :: B

- Notice that function application associates more tightly than any infix operator:

  f x + y = (f x) + y

- In types, arrows associate to the right:

  A -> B -> C = A -> (B -> C)

  Example: take :: Int -> [a] -> [a]

  take 2 [1,2,3,4] = (take 2) [1,2,3,4]

Sections:

- If ⊕ is a binary op of type A -> B -> C, then we can use "sections":
  - (⊕)   :: A -> B -> C
  - (expr ⊕) :: B -> C   (assuming expr::A)
  - (⊕ expr) :: A -> C   (assuming expr::B)

- Examples:
  - (1+), (2*), (1/), (<10), …

Higher-order Functions:

- map :: (a -> b) -> [a] -> [b]
  - map (1+) [1..5] = [2,3,4,5,6]

- takeWhile :: (a -> Bool) -> [a] -> [a]
  - takeWhile (<5) [1..10] = [1,2,3,4]

- (.) :: (a -> b) -> (c -> a) -> c -> b
  - (odd . (1+)) 2 = True

"composition"

Definitions:

- So far, we've been focusing on expressions that we might want to evaluate.

- What if we wanted to:
  - Define a new constant (i.e., Give a name to the result of an expression)?
  - Define a new function?
  - Define a new type?

- Definitions are placed in files with a .hs suffix that can be loaded into the interpreter

Simple Definitions:

Put the following text in a file "defs.hs":

```
greet name = "hello " ++ name

square x = x * x

fact n = product [1..n]
```

Loading Defined Values:

Pass the filename as a command line argument to Hugs, or use the :l command from inside Hugs:

```
Main> :l defs
Main> greet "everybody"
"hello everybody"
Main> square 12
144
Main> fact 32
263130836933693530167218012160000000
Main>
```

Using Libraries:

- Many useful functions are provided as part of the "Prelude"
- Many more are provided by libraries that must be imported before they can be used
- Example:

```
import Char
nextChar c = chr (1 + ord c)
```

- (The Char library also provides functions for converting to upper/lower case, testing for alphabetic or numeric chars, etc…)

Typeful Programming:

- Types are an inescapable feature of programming in Haskell
  - Programs, definitions, and expressions that do not type check are not valid Haskell programs
  - Compilation of Haskell code depends on information that is obtained by type checking
- Haskell provides several predefined types:
  - Some built-in (functions, numeric types, …)
  - Some defined in the Prelude (Bool, lists, …)
- What if you need a type that isn't built-in?

Type Synonyms:

Type Synonym:

- A type synonym (or type abbreviation) gives a new name for an existing type.
- Examples:

```
type String  = [Char]
type Length  = Float
type Angle   = Float
type Radius  = Length
type Point   = (Float, Float)
type Set a   = a -> Bool
```

Algebraic Datatypes:

In Haskell Notation:

data Bool = False | True
introduces:

- A type, Bool
- A constructor function, False :: Bool
- A constructor function, True :: Bool

data List a = Nil | Cons a (List a)
introduces

- A type, List t, for each type t
- A constructor function, Nil :: List a
- A constructor function, Cons :: a -> List a -> List a

More Enumerations:

```
data Rainbow  = Red | Orange | Yellow
              | Green | Blue | Indigo | Violet
```

introduces:

- A type, Rainbow
- A constructor function, Red :: Rainbow
- …
- A constructor function, Violet :: Rainbow

More Recursive Types:

```
data Shape      = Circle Radius
                 | Rect Length Length
                 | Transform Transform Shape
data Transform
                = Translate Point
                 | Rotate Angle
                 | Compose Transform Transform
```

introduces:

- Two types, Shape and Transform
- Circle :: Radius -> Shape
- Rect :: Length -> Length -> Shape
- Transform :: Transform -> Shape -> Shape
- …

Using New Data Types:

- Building values of these new types is easy:

```
Nil                          :: List Rainbow
Cons Red Nil                 :: List Rainbow
Cons Blue (Cons Red Nil)     :: List Rainbow
```

- But how do we inspect them or take them apart?

Pattern Matching:

- In addition to introducing a new type and a collection of constructor functions, each data definition also adds the ability to pattern match over values of the new type
- Example:

```
first                :: (a, b) -> a
first (x, y)         = x

wavelengths          :: Rainbow -> (Length,Length)
wavelengths Red      = (620*nm, 750*nm)
wavelengths Orange = (590*nm, 620*nm)
…
nm = 1e-9 :: Float
```

More Examples:

```
head                      :: [a] -> a
head []                   = error "head of []"
head (x:xs)               = x

length                    :: [a] -> Int
length []                 = 0
length (x:xs)             = 1 + length xs

area                      :: Shape -> Float
area (Circle r)           = pi * r * r
area (Rect w h)           = w * h
area (Transform t s)      = area s
```

Pattern Matching & Substitution:

- The result of a pattern match is either:
  - A failure
  - A success, accompanied by a substitution that provides a value for each of the values in the pattern

- Examples:
  - [] does not match the pattern (x:xs)
  - [1,2,3] matches the pattern (x:xs) with x=1 and xs=[2,3]

Patterns:

More formally, a pattern is either:

- An identifier
  - Matches any value, binds result to the identifier

- An underscore (a "wildcard")
  - Matches any value, discards the result

- A constructed pattern of the form $C\ p_1 \ldots p_n$, where C is a constructor of arity n and $p_1, \ldots, p_n$ are patterns of the appropriate type
  - Matches any value of the form $C\ e_1 \ldots e_n$, provided that each of the e_i values matches the corresponding p_i pattern.

Other Pattern Forms:

For completeness:

- "Sugared" constructor patterns:
  - Tuple patterns (p_1,p_2)
  - Cons patterns (ph : pt)
  - List patterns $[p_1, p_2, p_3]$
  - Strings, for example: "hi" = ('h' : 'i' : [])
- Character and numeric Literals:
  - Can be considered as constructor patterns, but the implementation uses equality (==) to test for matches

Function Definitions:

- In general, a function definition is written as a list of adjacent equations of the form:

  $$f\ p_1 \ldots p_n = rhs$$

  where:
  - f is the name of the function that is being defined
  - $p_1, \ldots, p_n$ are patterns, and rhs is an expression

- All equations in the definition of f must have the same number of arguments (the "arity" of f)

… continued:

- Given a function definition with m equations:

  $f\ p_{1,1}\ \ldots\ p_{n,1} = rhs_1$

  $f\ p_{1,2}\ \ldots\ p_{n,2} = rhs_2$

  …

  $f\ p_{1,m}\ \ldots\ p_{n,m} = rhs_m$

- The value of $f\ e_1\ \ldots\ e_n$ is $S\ rhs_i$, where i is the smallest integer such that the expressions e_j match the patterns $p_{j,i}$ and S is the corresponding substitution.

Example: filter

```
filter              :: (a -> Bool) -> [a] -> [a]
filter p []         = []
filter p (x:xs)
    | p x           = x : rest
    | otherwise     = rest
    where rest = filter p xs
```

guards

"where" clause

Example: Binary Search Trees

```
data Tree              = Leaf | Fork Tree Int Tree

insert                 :: Int -> Tree -> Tree
insert n Leaf          = Fork Leaf n Leaf
insert n (Fork l m r)
        | n <= m       = Fork (insert n l) m r
        | otherwise    = Fork l m (insert n r)

lookup                 :: Int -> Tree -> Bool
lookup n Leaf          = False
lookup n (Fork l m r)
      | n < m          = lookup n l
      | n > m          = lookup n r
      | otherwise      = True
```

Summary:

- An appealing, high-level approach to program construction in which independent aspects of program behavior are neatly separated
- It is possible to program in a similar compositional / calculational manner in other languages …
- … but it seems particularly natural in a functional language like Haskell …

Assignment #1

- Your goal is to write a function:
  - toInt :: String -> Int
- To accomplish this, consider the following functions:
  - explode :: String -> [Char]
  - digitValue :: [Char] -> [Int]
  - reverse :: [Int] -> [Int]
  - pairedWithPowersOf10 :: [Int] -> [(Int,Int)]
  - pairwiseProduct :: [(Int,Int)] -> [Int]
  - sum :: [Int] -> Int
- Write definitions for four of these functions (reverse and sum are built-in), using pattern matching and recursion where necessary
- Turn in an elegant program that communicates your solution well, including appropriate tests for each part.