## FUNCTIONAL PROGRAMMING IS PROGRAMMING WITHOUT...

- ...selective assignments (bad: `a[i] = 6`).
  - The goal of an imperative program is to change the state [of the machine].
  - The goal of a functional programs is to evaluate (reduce, simplify) expressions.
- ...in general, updating assignments (`y = x + 1` good; `x = x + 1` bad):
  - A variable in an imperative program: a name for a container.
  - There is no proper concept of "variable" in functional programs. What is called "variable" is a name for an expression.
- ...explicit pointers, storage management.
- ...input/output.
- ...control structures (loops, conditional statements).
- ...jumps (break, goto, exceptions).



## WHAT'S LEFT?

- Expressions (without side effects).
  - Referential transparency (i.e., substitutivity, congruence).
- Definitions (of constants, functions).
  - Functions (almost as in mathematics).

| Math | Haskell |
|---|---|
| square | `square` |

- Types (including higher-order, polymorphic, and recursively-defined types).
  - tuples, lists, and trees, shared sub-structures, implicit cycles.
- Automatic storage management (garbage collection).



## WHAT'S LEFT?

- Expressions (without side effects).
  - Referential transparency (i.e., substitutivity, congruence).
- Definitions (of constants, functions).
  - Functions (almost as in mathematics).

| Math | Haskell |
|---|---|
| square : $\mathbb{N} \rightarrow \mathbb{N}$ | `square` |
| square$(x) = x \times x$ | |

- Types (including higher-order, polymorphic, and recursively-defined types).
  - tuples, lists, and trees, shared sub-structures, implicit cycles.
- Automatic storage management (garbage collection).



## WHAT'S LEFT?

- Expressions (without side effects).
  - Referential transparency (i.e., substitutivity, congruence).
- Definitions (of constants, functions).
  - Functions (almost as in mathematics).

| Math | Haskell |
|---|---|
| square : $\mathbb{N} \rightarrow \mathbb{N}$ | `square :: Integer -> Integer` |
| square$(x) = x \times x$ | `square x = x * x` |

  A function is defined by a set of rewriting rules.

- Types (including higher-order, polymorphic, and recursively-defined types).
  - tuples, lists, and trees, shared sub-structures, implicit cycles.
- Automatic storage management (garbage collection).

## SESSIONS, SCRIPTS, EVALUATION

```
< godel:306/slides > ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 66
66
Prelude> 6 * 7
42
Prelude> square 35567
<interactive>:4:1: Not in scope:
'square'
Prelude> :load example
[1 of 1] Compiling Main
( example.hs, interpreted )
Ok, modules loaded: Main.
*Main> square 35567
1265011489
*Main> square (smaller (5, 78))
25
*Main> square (smaller (5*10, 5+10))
225
*Main>
```

(file example.hs)

```
-- a value (of type Integer):

infty :: Integer
infty = infty + 1

-- a function
-- (from Integer to Integer):

square :: Integer -> Integer
square x = x * x

-- another function:

smaller :: (Integer,Integer)->Integer
smaller (x,y) = if x<=y then x else y
```



## WHAT'S LEFT? (CONT'D)

- Functions are first order objects.

```
twice :: (Integer -> Integer) -> (Integer -> Integer)
twice f = g
    where g x = f (f x)
```

- A program (or script) is a collection of definitions.
- Predefined data types in a nutshell:
  - Numerical: $Integer$, $Int$, $Float$, $Double$.
  - Logical: $Bool$ (values: $True$, $False$).
  - Characters: $Char$ ('a', 'b', etc.).
  - Composite: Functional: $Integer \to Integer$;
    Tuples: $(Int, Int, Float)$;
    Combinations: $(Int, Float) \to (Float, Bool)$, $Int \to (Int \to Int)$.



## THINK RECURSIVELY

**Instructions for reading a book:**

- C: "While not on the end cover repeat: read the current page, set the current page to the next page."
- Functional: "If on the end cover, stop. Otherwise, read the first page, then read recursively the rest of the book."

- Other examples:
  - To climb a ladder, step on the first rung and then climb (recursively) the rest of the ladder.
  - To eat a six-course meal, eat the first meal and then eat (recursively) the rest of the meal.
- How does one compute the factorial on a number?



## SCRIPTS

- Recall that a program is a collection of definitions of values (including functions).
- Syntactical sugar: definitions by guarded equations:

```
smaller :: (Integer, Integer) -> Integer
smaller (x,y)
   | x <= y  = x
   | x > y   = y
```

- Recursive definitions:

```
fact   :: Integer -> Integer
fact x = if x==0 then 1 else x * fact (x-1)
```

- Syntactical sugar: definitions by pattern matching (aka by cases):

```
fact   :: Integer -> Integer
fact 0 = 1
fact x = x * fact (x-1)
```

## LOCAL DEFINITIONS

- Two forms:

```
let v1 = e1                def
    v2 = e2                  where v1 = e1
    .                              v2 = e2
    .                              .
    .                              .
    vk = ek                        .
in exp                             vk = ek
```

- Definitions are qualified by **where** clauses, while expressions are qualified by **let** clauses.



## SCOPING

- Haskell uses static scoping.

```
cylinderArea :: Float -> Float -> Float
cylinderArea h r = h * 2 * pi * r + 2 * pi * r * r

cylinderArea1 :: Float -> Float -> Float
cylinderArea1 h r = x + 2 * y
    where x = h * circLength r
          y = circArea r
          circArea x = pi * x * x
          circLength x = 2 * pi * x

cylinderArea2 :: Float -> Float -> Float
cylinderArea2 h r = let x = h * circLength r
                        y = circArea r
                        in x + 2 * y
    where circArea x = pi * x * x
          circLength x = 2 * pi * x
```



## TYPES

- Each type has associated operations that are not necessarily meaningful to other types.
  - Arithmetic operations ($+$, $-$, $*$, $/$) can be applied to numerical types, but it does not make any sense to apply them on, say, values of type $Bool$.
  - It does, however make sense to compare (using $=$ (==), $\neq$ (/=), $\leq$ (<=), $<$, etc.) both numbers and boolean values.
- Every well formed expression can be assigned a type (strong typing).
  - the type of an expression can be inferred from the types of the constituents of that expression.
  - those expression whose type cannot be inferred are rejected by the compiler.

```
badType x                   fact :: Integer -> Integer
  | x == 0 = 0              fact x
  | x > 0  = 'p'              | x < 0  = error "Negative argument."
  | x < 0  = 'n'              | x == 0 = 1
                              | x > 0  = x * fact (x-1)
```

What is the type of `error`?



## TWO DATA TYPES

- **Booleans**. Values: $True$, $False$.
  - operations on $Bool$: logic operators: $\vee$ (||), $\wedge$ (&&), $\neg$ (not); comparisons: $=$ (==), $\neq$ (/=); relational $<$, $\leq$ (<=), $>$, $\geq$ (>=).
- **Characters**. Values: 256 of them, e.g., 'a', 'b', '\n'.
  - Oerations on characters: comparison, relational;

```
ord :: Char -> Int          | Prelude> import Data.Char
chr :: Int -> Char          | Prelude Data.Char> ord 'a'
                            | 97
                            | Prelude Data.Char> chr 100
                            | 'd'
toLower :: Char -> Char
toLower c | isUpper c = chr (ord c - (ord 'A' - ord 'a'))
          | True      = c
    where isUpper c = 'A' <= c && c <= 'Z'
```

## Lists

- A list is an ordered set of values.

| | | |
|---|---|---|
| $[1, 2, 3] :: [Int]$ | $[[1, 2], [3]] :: [[Int]]$ | $['h', 'i'] :: [Char]$ |
| $[div, rem] ::\ ??$ | $[1, 'h'] ::\ ??$ | $[] ::\ ??$ |

- Syntactical sugar:

```
Prelude> ['h','i']
"hi"
Prelude> "hi" == ['h','i']
True
Prelude> [['h','i'],"there"]
["hi","there"]
```



## Constructing lists

- Constructors: [] (the empty list) and : (constructs a longer list).

```
Prelude> 1:[2,3,4]
[1,2,3,4]
Prelude> 'h':'i':[]
"hi"
```

  - The operator ":" (pronounced "cons") is right associative.
  - The operator ":" **does not** concatenate lists together!

```
Prelude> [1,2,3] : [4,5]
    No instance for (Num [t0])
      arising from the literal '4'
    Possible fix: add an instance declaration for (Num [t0])
    In the expression: 4
    In the second argument of '(:)', namely '[4, 5]'
    In the expression: [1, 2, 3] : [4, 5]
Prelude> [1,2,3] : [[4,5]]
[[1,2,3],[4,5]]
Prelude> [1,2,3] ++ [4,5]
[1,2,3,4,5]
Prelude>
```



## Operations and pattern matching on lists

- Comparisons ($<$, $\geq$, $==$, etc.), if possible, are made in lexicographical order.
- Subscript operator: !! (e.g., $[1, 2, 3]$ !! 1 evaluates to 2) – expensive
- Arguably the most common list processing: Given a list, do something with each and every element of that list.
  - In fact, such a processing is so common that there exists the predefined $map$ that does precisely this:

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

  - This is also an example of pattern matching on lists.
    - Variant to pattern matching: $head$ and $tail$ (predefined).

```
head (x:xs) = x     | map f l = if l == []
tail (x:xs) = xs    |             then []
                    |             else f (head l) : map f (tail l)
```



## Tuples

- While lists are homogenous, tuples group values of (posibly) diferent types.

```
divRem :: Integer -> Integer -> (Integer, Integer)
divRem x y = (div x y, rem x y)

divRem1 :: (Integer, Integer) -> (Integer, Integer)
divRem1 (x, 0) = (0, 0)
divRem1 (x, y) = (div x y, rem x y)
```

- The latter variant is also an example of pattern matching on tuples.

## Operators and functions

- An operator contains symbols from the set !#$%&*+./<=>?@\^|: (− and ˜ may also appear, but only as the first character).
- Some operators are predefined (+, −, etc.), but you can define your own as well.
- An (infix) operator becomes (prefix) function if surrounded by brackets. A (prefix) function becomes operator if surrounded by backquotes:

```
divRem :: Integer -> Integer -> (Integer, Integer)
x ‘divRem‘ y = (div x y, rem x y)
-- precisely equivalent to
-- divRem x y = (div x y, rem x y)

(%%) :: Integer -> Integer -> (Integer, Integer)
(%%) x y = (div x y, rem x y)
-- precisely equivalent to
-- x %% y = (div x y, rem x y)
```

```
Main> 3 %% 2
(1,1)
Main> (%%) 3 2
(1,1)
Main> divRem 3 2
(1,1)
Main> 3 ‘divRem‘ 2
(1,1)
Main>
```

These are just lexical conventions.



## Identifiers

- Identifiers consist in letters, numbers, simple quotes ('), and underscores (_), but they must start with a letter.
  - For the time being, they must actually start with a lower case letter.
    - A Haskell idenitifer starting with a capital letter is considered a type (e.g., $Bool$) or a type constructor (e.g., $True$)—we shall talk at length about those later.
    - By convention, types (i.e., class names) in Java start with capital letters, and functions (i.e., method names) start with a lower case letter. What is a convention in Java is the rule in Haskell!
  - Some identifiers are language keywords and cannot be redefined (`if`, `then`, `else`, `let`, `where`, etc.).
    - Some identifiers (e.g., `either`) are defined in the standard prelude and possibly cannot be redefined (depending on implementation, messages like “`Definition of variable "either" clashes with import`”).