Haskell: yet another Functional Language

Haskell is a polymorphically-typed, lazy, purely-functional language.
Hence Haskell is similar to SML, e.g.

SML: Haskell:

```haskell
fun fac 1 = 1 | fac 1 = 1
fac n = n*fac (n-1); fac n = n*fac (n-1)
> val fac = fn : int -> int
- fac 4;  *Main> fac 4
> 24 : int 24
```

Referential Transparency

**Definition** (Stoy,1977): The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same wherever it occurs.

Other definitions & discussion
(Whitehead&Russell,1925) (Quine,1960)

Implications:

- Two expressions are *equal* if they have the same value, e.g. \( \sin(6) = \sin(1+5) \).
- Value-based equality enables *equational reasoning*, where one expression is substituted by another of equal value, e.g. \( f(x) + f(x) = 2f(x) \).
- Scope matters: if \( x = 6 \), then \( \sin(x) = \sin(6) = \sin(1+5) \).

Characteristics of Functional Languages

Like other modern functional languages e.g. SML or Scheme, Haskell includes advanced features:

- Sophisticated polymorphic type system, with type inference.
- Pattern matching.
- Higher-order functions.
- Data abstraction.
- Garbage-collected storage management.

These features are also found in modern procedural and O-O languages. The distinctive feature of pure functional languages is their referential transparency.
**Referentially Opaque Notations**

**English:**
“Eric the Red was so called because of his red beard”

Cannot substitute “Eric Jarlsson” for “Eric the Red” and retain the meaning.

**Procedural programming languages:**
\[ x = x + 1 \]

**Exercise:** In C, does replacing the sum of two function calls: \( f(x) + f(x) \), by \( 2f(x) \) preserve the meaning of the program? If not, give an example function, \( f \) that should not be substituted.

---

**Referentially Transparent Notations**

Most of mathematics is referentially transparent.

- integer algebras: \( 2x + x = 12 \).
- relational algebra: \( R \cup S = S \cup R \)

As SQL is based on the relational algebra, a large subset of it is referentially transparent, except for parts that change the database, e.g. UPDATE, DELETE etc.

**Mathematical logics:** \( P(x) \land Q(x) \). As a result, a large part of most logic, or deductive, languages is referentially transparent, except for features like CUT, ASSERT and RETRACT.

Purely functional languages

---

**Consequences of Referential Transparency**

**Equational reasoning**
- Proofs of correctness are much easier than reasoning about state as in procedural languages.
- Used to transform programs, e.g. to transform simple specifications into efficient programs.

**Freedom from execution order.**
- Meaning of program is not dependent on execution order: so programs are shorter as the programmer doesn’t have to specify an execution order.
- Lazy evaluation Most languages have a strict evaluation order, e.g. evaluate the parameters to a function left-to-right before calling it.

A lazy language only evaluates an expression when, and if, it’s needed.

- *Parallel/distributed evaluation.* Often there are many expressions that can be evaluated at a time, because we know that the order of evaluation doesn’t change the meaning, the sub-expressions can be evaluated in parallel (Wegner 1978)

**Elimination of side effects.** A side effect is an unexpected action performed by a sub-program, e.g. assigning to a global variable.
Downside of referential transparency

Interaction with state outside the program, e.g. reading from a file, updating a database is harder.
⇒ language needs special constructs for dealing with stateful objects, and Haskell uses Monads

Differences between SML and Haskell

Where SML is an imperative language with a large purely-functional subset, Haskell is (almost entirely) purely-functional, guaranteeing referential transparency.
Where SML is strict with a defined evaluation order, Haskell is lazy.
Other minor differences, e.g.s: different module systems, Haskell has parametric polymorphism (Java 1.5’s generics are based on it), etc.

1. Haskell Types & Values

All Haskell values are first-class - they may be passed as arguments to functions, returned as results, placed in data structures, etc.

Example Haskell values and types:

5 :: Integer
’a’ :: Char
True :: Bool
inc :: Integer -> Integer
[1,2,3] :: [Integer]
(’b’,4) :: (Char,Integer)

Function Definitions

Functions are normally defined by a series of equations. Giving type signatures for functions is optional, but highly recommended.

inc :: Integer -> Integer
inc n = n+1

To indicate that an expression e₁ evaluates to another expression e₂, we write e₁ => e₂, e.g.

inc (inc 3) => 5

Function definition equations often use pattern matching, e.g.

fac :: Integer -> Integer
fac 1 = 1
fac n = n*fac (n-1)
Lists

Constructed from cons (:) and nil ([]), e.g.

1:[]
1:2:3:[]
'b':'y':'e':[]

having types [Integer], [Integer] and [Char].

Lists are commonly abbreviated:

[1]
[1,2,3]
['b','y','e']

A list can be indexed with the !! operator:

[1,2,3] !! 0 => 1
['b','y','e'] !! 2 => 'e'

A list can be enumerated:

[1 .. 5] => [1,2,3,4,5]

List Comprehensions

Sets are often written in mathematics as comprehensions, e.g.

\{x | x \in \{1..10\} \land \text{odd}(x)\}

Haskell has analogous list comprehensions:

[ x | x <- [1 .. 10], odd x]
=>
[1,3,5,7,9]

There may be many generators or filters, e.g. cartesian product:

[ (x,y) | x <- ['a','b'], y <- [3,4] ]
=>
[('a',3),('a',4),('b',3),('b',4)]

Polymorphic Functions

A polymorphic function (generic method in Java 1.5) can operate on values of many types, e.g.

\text{length} :: [a] \to \text{Integer}

length [] = 0
length (x:xs) = 1 + length xs

length [1,2,3] => 3
length ['b','y','e'] => 3
length [[1],[2]] => 2

a is a type variable (like 'a in SML) and must be lower case in Haskell.
User-defined Types

Data constructors are introduced with the keyword `data`.

Nullary data constructors, or enumerated types:

```haskell
data Bool = False | True
data Color = Red | Green | Blue
```

A polymorphic unary data constructor:

```haskell
data Point a = Pt a a
```

```haskell
Pt 2.0 3.0 :: Point Float
Pt 'a' 'b' :: Point Char
Pt True False :: Point Bool
```

A recursive data constructor:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

```haskell
fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Here `++` is the infix operator that concatenates two lists.

N.B. type constructor names must be capitalised.

Type Synonyms

Type synonyms are names for commonly used types, rather than new types, and defined with the keyword `type`:

```haskell
type String = [Char]
type Person = (Name, Address)
type Name = String
data Address = None | Addr String
```

Syntactic support is provided for strings, e.g. `'bye'` => `['b','y','e']`, and list operations can be applied to them, e.g. `length 'bye'` => 3.

2. Haskell Functions

Pattern Matching

A pattern may contain a wildcard, e.g. to chose just the first `n` elements of a list, `take 2 [1,2,3] => [1,2]`

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

**Exercise:** What does `take -1 [1,2,3,4]` reduce to?
Guarded Patterns

A pattern may contain a **guard**: a condition that must be true for the pattern to match, e.g.

```
sign x | x > 0 = 1
| x == 0 = 0
| x < 0 = -1
```

Local Definitions

Haskell, like SML has a mutually recursive **let** binding, e.g.

```
let y = a*b
    f x = (x+y)/y
in f c + f d
```

The Haskell **where** binding scopes over several guarded equations, e.g.

```
f x y | y>z = ...
     | y==z = ...
     | y<z = ...
     where z = x*x
```

Layout

Haskell lays out equations in columns to disambiguate between multiple equations, e.g. could previous definition be:

```
let y = a*b f
    x = (x+y)/y
in f c + f d
```

Key rule: declarations start to the right of **where** or **let**.

Curried & Infix Functions

**Currying**: a function requiring n arguments can be applied to fewer arguments to get another function, e.g.

```
add x y = x + y
```

```
inc :: Integer -> Integer
inc = add 1
```

A binary function can be written **infix** using backquote, e.g.

```
3 `add` 4 => 7
```
Higher Order Functions

Functions are first class values and can be passed as arguments to other functions and returned as the result of a function.

Many useful higher-order functions are defined in the Prelude and libraries, including most of those from your SML course, e.g.

- **filter** takes a list and a boolean function and produces a list containing only those elements that return `True`

  ```haskell
  filter :: (a -> Bool) -> [a] -> [a]
  filter p [] = []
  filter p (x:xs) =
  | p x = x: filter p xs
  | otherwise = filter p xs
  
  filter odd [1,2,3]
  => [1,3]
  
  - **map** applies a function `f` to every element of a list

    ```haskell
    map :: (a -> b) -> [a] -> [b]
    map f [] = []
    map f (x:xs) = (f x): map f xs
    
    map inc [2,3,4]
    => (inc 2) : map inc [3,4]
    => (inc 2) : (inc 3) : map inc [4]
    => (inc 2) : (inc 3) : (inc 4) : map inc []
    => (inc 2) : (inc 3) : (inc 4) : []
    ...
    => [3,4,5]
    
    - **foldr** applies a binary function to every element of a list:

      ```haskell
      foldr :: (a -> b -> b) -> b -> [a] -> b
      foldr f z [] = z
      foldr f z (x:xs) = f x (foldr f z xs)
      
      foldr add 0 [2,3,4]
      => add 2 (foldr add 0 [3,4])
      => add 2 (add 3 (foldr add 0 [4]))
      => add 2 (add 3 (add 4 (foldr add 0 [])))
      => add 2 (add 3 (add 4 0))
      ...
      => 9
      
      - **zip** function converts a pair of lists into a list of pairs:

        ```haskell
        zip :: [a] -> [b] -> [(a,b)]
        
        zip [1,2,3] [9,8,7]
        => [(1,9),(2,8),(3,7)]
        
        - **zipWith** takes a pair of lists and a binary function and produces a list containing the result of applying the function to each ‘matching’ pair:

          ```haskell
          zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
          
          zipWith (+) [1,2,3] [3,2,1]
          => [4,4,4]
          ```
Lambda Abstractions

Functions may be defined “anonymously” via a lambda abstraction (\fn in SML). For example definitions like

\[\text{inc } x = x + 1\]
\[\text{add } x \ y = x + y\]

are really shorthand for:

\[\text{inc } = \lambda x . x + 1\]
\[\text{add } = \lambda x \ y . x + y\]

Infix Operators

Infix operators are really just functions, and can also be defined using equations, e.g. list concatenation:

\[\text{[ ] ++ } y s = y s\]
\[\text{(x:xs) ++ } y s = x \ : \ (x s ++ y s)\]

and function composition:

\[\text{. } : : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)\]
\[f \ . \ g = \lambda x . f (g x)\]

Lexically, infix operators consist entirely of "symbols," as opposed to normal identifiers which are alphanumeric.

Sections

As operators functions they can be curried, e.g. (parentheses mandatory)

\[\text{(+)} = \lambda y . x + y\]
\[\text{(y+)} = \lambda x . x + y\]
\[\text{(+) = } \lambda x \ y . x + y\]

Lazy Functions

Most programming languages have strict semantics: the arguments to a function are evaluated before the evaluating the function body. This sometimes wastes work, e.g.

\[f \ True \ y = 1\]
\[f \ False \ y = y\]

It may even cause a program to fail when it could complete, e.g.

\[f \ True \ (1/0)\]

Haskell functions are lazy: the evaluation of the arguments is delayed, and the body of the function is evaluated and only if the argument is actually needed (or demanded) is it evaluated.
“Infinite” Data Structures

Data constructors are also lazy (they’re just a special kind of function), e.g. list construction (`:`)

Non-strict constructors permit the definition of (conceptually) infinite data structures. Here is an infinite list of ones:

\[
\text{ones} = 1 : \text{ones}
\]

More interesting examples, successive integers, and all squares (using infix exponentiation):

\[
\text{numsFrom } n = n : \text{numsFrom } (n+1) \\
\text{squares} = \text{map } (^2) \text{ (numsfrom 0)}
\]

Any program only constructs a finite part of an infinite sequence, e.g.

\[
\text{take 5 squares} => [0,1,4,9,16]
\]

Normal Forms

Normal forms are defined in terms of reducable expressions, or redexes, i.e. expressions that can be simplified e.g. \((+ \ 3 \ 4)\) is reducable, but \(7\) is not.

Strict languages like SML reduce expressions to Normal Form (NF), i.e. until no redexes exist. Example NF expressions:

\[
5 \quad [4,5,6] \quad \lambda x \to x + 1
\]

Lazy languages like Haskell reduce expressions to Weak Head Normal Form (WHNF), i.e. until no top level redexes exist. Example non-WHNF expressions:

\[
(+ \ 4 \ 1) \quad (\lambda x \to x + 1 \ 3)
\]

3. Type Classes and Overloading

In addition to the parametric polymorphism already discussed, e.g. 

\[
\text{length :: } [\text{a}] \to \text{Int}
\]

Haskell also supports ad hoc polymorphism or overloading, e.g. 

- 1, 2, etc. represent both fixed and arbitrary precision integers.
- Operators such as + are defined to work on many different kinds of numbers.
- Equality operator (==) works on numbers and other types.

Note that these overloaded behaviors are different for each type and may be an error, whereas in parametric polymorphism the type truly does not matter, e.g. \text{length} works for lists of any type.
Declaring Classes and Instances

It is useful to define equality for many types, e.g. String, Char, Int, etc, but not all, e.g. functions.

A Haskell class declaration, with a single method:

```
class Eq a where
    (==) :: a -> a -> Bool
```

Example instance declarations, integerEq and floatEq are primitive functions:

```
instance Eq Integer where
    x == y = x 'integerEq' y

instance Eq Float where
    x == y = x 'floatEq' y

instance (Eq a) => Eq (Tree a) where
    Leaf a == Leaf b = a == b
    (Branch l1 r1) == (Branch l2 r2) = (l1==l2)
    _ == _ = False
```

Number Classes

Haskell has a rich set of numeric types and classes that inherit in the obvious ways. The root of the numeric class hierarchy is Num.

- **Integer** in class Integral: Arbitrary-precision integers
- **Int** in class Integral: Fixed-precision integers
- **Float** in class RealFloat: Real floating-point, single precision

Input/Output

To preserve referential transparency, stateful interaction in Haskell (e.g. I/O) is performed in a Monad.

Input/Output actions occur in the IO Monad, e.g.

```
getChar :: IO Char
putChar :: Char -> IO ()

getArgs :: IO [String]
putStrLn, putStrLn :: String -> IO ()
```

Every Haskell program has a main :: IO () function, e.g.

```
main = putStrLn "Hello"
```

A do statement performs a sequence of actions, e.g.

```
main :: IO ()
main = do c <- getChar
        putChar c
```

Useful I/O

Many useful IO functions are in the system module and must be imported, see below.

- **show** :: (Show a) => a -> String converts values of most types into a String, e.g. `show 3 => "3"`
- **read** :: (Read a) => String -> a parses a value of most types from a String. Explicit type information may be required to disambiguate overloaded values, e.g.

```
read "True":Bool => True
read "(True,3)":: (Bool,Int) => (True,3)
```

A program returning the sum of it’s command line arguments:

```
import System

main = do args <- getArgs
        let x = read (args!!0)
        let y = read (args!!1)
        putStrLn (show (x+y))
```
Haskell Pragmatics

Many useful Haskell functions are available:

- the standard prelude available in every program.
  http://www.cs.uu.nl/~afie/haskell/tourofprelude.html

- standard libraries modules that must be imported.
  http://www.haskell.org/onlinelibrary/

- other libraries.
  http://www.haskell.org/libraries/

Uses of Haskell

Rapid prototype development

Symbolic applications development, e.g. natural language processors, chess games.

Teaching High-level programming

Computation language for parallel, distributed, mobile or Grid languages