Features

- Named after logician Haskell B. Curry (after whom the currying transformation is named).
- Pure functional language (imperative features like I/O integrated into a functional framework).
- Infix syntax.
- Polymorphic static typing.
- Equational definitions.
- Pattern-matching.
- Lazy evaluation (normal-order reduction).
- First-class functions.

Currying

- Given a function f of type $f : (X \times Y) \to Z$, then $\text{curry}(f) : X \to (Y \to Z)$.
- Currying results in higher-order functions.

Data

- Numbers, including unlimited precision integers.
- Characters. `'a'`, `'b'`, ...
- Strings `"abc"` equivalent to list of characters.
- Lists. List literals enclosed within square brackets. Infix `:` used as list constructor. Empty list denoted as `[]`.
  `[1, 2, 3]` equivalent to `1 : (2 : (3 : []))`.
- Tuples.

Factorial Program

In factorial.hs

```
factorial 0 = 1
factorial n = n * factorial(n - 1)
```

Factorial Log

```
Main> :l "factorial.hs"
Reading file "factorial.hs":

Hugs session for:
/usr/local/lib/hugs/lib/Prelude.hs
factorial.hs
Main> factorial 0
1
Main> factorial 4
24
Main> :t factorial
factorial :: Num a => a -> a
Main> factorial 3.0
6.0
Main>
```

List Length Program

In my-length.hs

```
--optional type declaration.
my_length :: [a] -> Int

my_length [] = 0
my_length (x:xs) = 1 + my_length xs
```

List Length Log

```
Prelude> :l "my_length.hs"
...
Main> my_length []
0
Main> my_length [1, 2, 3]
3
Main> my_length 1
ERROR - Unresolved overloading
*** Type       : Num [a] => Int
*** Expression : my_length 1

Main> my_length ["a"]
1
Main>
```

Can contain non-uniform types.

```
Main> (1, 2, 3)
(1,2,3)
Main> :t (1, 2, 3)
(1,2,3) :: (Num a, Num b, Num c) => (c,b,a)
Main> :t (1, 2, 'a')
(1,2,'a') :: (Num a, Num b) => (b,a,Char)
Main> (1, 2, 'a')
(1,2,'a')
Main> fst (1, "a")
1
Main> snd (1, "a")
"a"
Main>
```

lambda-length.hs

```
lambda_length =
  \ ls -> if null ls
     then 0
      else 1 + lambda_length(tail ls)
```

```
Main> :l "factorial.hs"
Reading file "factorial.hs":

Hugs session for:
/usr/local/lib/hugs/lib/Prelude.hs
factorial.hs
Main> factorial 0
1
Main> factorial 4
24
Main> :t factorial
factorial :: Num a => a -> a
Main> factorial 3.0
6.0
Main>
```

Type Inference

```
Main> :t 12
12 :: Num a => a
Main> 12
12
Main> :t [1, 2, 3]
[1,2,3] :: Num a => [a]
Main> [1, 2, 3]
[1,2,3]
Main> :t ["a", "bc", "d"]
["a","bc","d"] :: [[Char]]
```

Type Inference Continued

```
Main> ["a", "bc", "d"]
["a","bc","d"]
Main> ["a", "1"]
["a","1"]
Main> ["a", 1]
ERROR - Unresolved overloading
*** Type       : Num [Char] => [[Char]]
*** Expression : ["a",1]

Main>
```

A string is a list of characters. ++ used for list and string concatenation.

```
Main> ['h', 'e', 'l', 'l', 'o' ]
"hello"
Main> ['h', 'e', 'l', 'l', 'o', ' ' ] ++ "world"
"hello world"
Main> length "hello"
5
```

Lists and Strings Continued

```
Main> [1, 2, 3] ++ [4, 5]
[1,2,3,4,5]
Main> [1, 2, 3] ++ ['a', 'b']
ERROR - Unresolved overloading
*** Type        : Num Char => [Char]
*** Expression : [1,2,3] ++ ['a','b']

Main>
```

List Comprehension

List comprehension returns a list of elements created by evaluation of generators Examples

```
Main> [x + 2*x + x/2 | x <- [1, 2, 3, 4]]
[3.5,7.0,10.5,14.0]
Main>  [ odd x | x <- [1..9]]
[True,False,True,False,True,False,True,False,True]
Main> [ x*y | x <- [1,2,3,4], y <- [3,5,7,9]]
[3,5,7,9,6,10,14,18,9,15,21,27,12,20,28,36]
Main> [x | x <- [1,5,12,3,23,11,7,2], x>10]
[12,23,11]
Main> [(x,y) | x <- [1,3,5], y <- [2,4,6], x<y]
[(1,2),(1,4),(1,6),(3,4),(3,6),(5,6)]
Main>
```

Quicksort

qsort.hs:

```
qsort []       = []
qsort (x:xs)  =
  qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
     where
       elts_lt_x    = [y | y <- xs, y < x]
       elts_greq_x = [y | y <- xs, y >= x]
```

Quicksort Log

```
Main> :l "qsort.hs"
...
Main> qsort [5, 3, 2, 6]
[2,3,5,6]
Main> qsort ["c", "d", "abc"]
["abc","c","d"]
Main> :t qsort
qsort :: Ord a => [a] -> [a]
Main>
```

Higher-Order Functions

```
Main> :t map
map :: (a -> b) -> [a] -> [b]
Main> map (\x -> x > 4) [1..9]
[False,False,False,False,True,True,True,True,True]
Main> map (\x -> x + 4) [1..9]
[5,6,7,8,9,10,11,12,13]
Main> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
Main> foldr (+) 0 [1, 2, 3, 4]
10
```

Higher-Order Functions Continued

```
Main> foldr (-) 0 [1, 2, 3, 4]
-2
Main> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
Main> foldl (+) 0 [1, 2, 3, 4]
10
Main> foldl (-) 0 [1, 2, 3, 4]
-10
Main> foldl (-) 15 [1, 2, 3, 4]
5
Main> foldl (-) (-15) [1, 2, 3, 4]
-25
```

```
Main> foldl (\x -> \y -> x + length y) 0
        ["abc", "d", "ef"]
6
Main> foldl (\x -> \y -> length x + y) 0
        ["abc", "d", "ef"]
ERROR - Type error in application
*** Expression     : foldl (\x -> \y -> length x + 
*** Term           : \x -> \y -> length x + y
*** Type           : [a] -> Int -> Int
*** Does not match : Int -> [Char] -> Int

Main>
```

Infinite Lists

infinite.hs

```
numsFrom :: Int -> [Int]
numsFrom n = n : numsFrom (n+1)

squares :: [Int]
squares = map (^2) (numsFrom 0)
```

Infinite Lists Log

```
Main> :t take
take :: Int -> [a] -> [a]
Main> take 5 (numsFrom 3)
[3,4,5,6,7]
Main> take 5 squares
[0,1,4,9,16]
Main>
```

Zip

`zip` is a function which turns two lists into a list of 2 tuples.
`zipWith` maps a binary function over two lists at once.

```
Main> zip [1, 2, 3] ["ab", "cd", "ef"]
[(1,"ab"),(2,"cd"),(3,"ef")]
Main> zip [1, 2, 3] ["ab", "cd"]
[(1,"ab"),(2,"cd")]
Main> zipWith (*) [1, 2, 3] [4, 5, 6]
[4,10,18]
Main> zipWith (*) [1, 2, 3] [4, 5]
[4,10]
Main>
```

Zip Fibonacci

zip-fib.hs

```
fib1 :: [Int]
fib1 =
  1 : 1 : [x+y | (x,y) <- zip fib1 (tail fib1)]

fib2 :: [Int]
fib2 =
  1 : 1 : map (\(x,y) -> x+y) (zip fib2 (tail fib2))

fib3 :: [Int]
fib3 = 1 : 1 : zipWith (+) fib3 (tail fib3)
```

Zip Fibonacci Log

```
Main> take 10 fib1
[1,1,2,3,5,8,13,21,34,55]
Main> take 10 fib2
[1,1,2,3,5,8,13,21,34,55]
Main> take 10 fib3
[1,1,2,3,5,8,13,21,34,55]
Main>
```

Case and Indentation

- Types start with upper-case letter; non-types with lower-case letter.
- Implicit semi-colon at the end of every line, except when continuation lines are indented.
- A semi-colon is inserted at EOF or whenever the next line starts in the left-hand margin.
- Blocks are indicated by indentation of keywords like where, let, of, do.
- New margin is indentation of token after keyword. Block ends at return to old margin.
- Can use explicit braces and semicolons.

Use of Guards

Guards can be used instead of a top-level `if` within an equation.

```
max :: Ord a => a -> a -> a
max x y
  | x > y = x
  | otherwise = y
```

Guards with Pattern Matching

Allows pattern matching within expressions:

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

Case Expressions

```haskell
my_length :: [a] -> Int
my_length xs =
  case xs of
    [] -> 0
    (x:xs) -> 1 + my_length xs
```

Function Composition

- Composition of two functions f and g is denoted using $f \,.\, g$. $(f \,.\, g)x = f(gx)$.
- Since function application has higher precedence than composition operator `.`, `succ . succ 1` is not `(succ . succ) 1)`, but `succ . (succ 1)`, which will usually result in a type error.
- Composition combines 2 functions, whereas application applies a function to a argument.

Typedefs

Define type synonyms using `type` keyword.

```
type Person = (String, String, int)
donald = ("Donald Duck", "123-45-6789", 63)
```

```
data DayOfWeek =
  Sun | Mon | Tue | Wed | Thu | Fri | Sat

data Shape
  = Circle (Float, Float) Float
  | Square (Float, Float) Float
  | Polygon [(Float, Float)]
```

Partial Computation

```haskell
data Maybe a
  = Nothing
  | Just a
```

- Can be used to convert a partial function to type `a` to a total function to `Maybe a`.
- `Nothing` used to indicate failure.
- `Just a` used to indicate success.

Partial Computation Example

In safe-division.hs:

```
safe_division :: Float -> Float -> Maybe Float
safe_division x y
  | y == 0 = Nothing
  | otherwise = Just (x/y)
```

In trees.hs:

```
data Tree a
  = Leaf a
  | InternalNode (Tree a) a (Tree a)

sumTree :: Tree Int -> Int
sumTree (Leaf value) = value
sumTree (InternalNode left v right) =
  sumTree left + v + sumTree right
```

Binary Trees Log

```
Main> sumTree (InternalNode (Leaf 1) 2 (Leaf 3))
6
(23 reductions, 45 cells)
Main> sumTree
        (InternalNode
          (Leaf 1) 2
          (InternalNode (Leaf 3) 4 (Leaf 5)))
15
(32 reductions, 31 cells)
Main>
```

User-Defined Operators

- Operator names consists of special characters

  `!#$%&*+./<=>?@\^|-~`

- Precedence declarations given by *fixity declarations* `infixr` *n* (right associative), `infixl` *n* (left associative), `infix` *n* (non-associative) for $n \in 1 \ldots 9$, with 9 being the strongest (function application has precedence level 10).

Stack Implementation

From distribution examples:

```haskell
-- Stacks: using restricted type synonyms

module Stack where

type Stack a =
     [a] in emptyStack, push, pop, topOf, isEmpty

emptyStack :: Stack a
emptyStack  = []

push        :: a -> Stack a -> Stack a
push         = (:)
```

Stack Implementation Continued

```
pop          :: Stack a -> Stack a
pop []        = error "pop: empty stack"
pop (_:xs)    = xs

topOf        :: Stack a -> a
topOf []      = error "topOf: empty stack"
topOf (x:_)   = x
```

Stack Implementation Continued

```
isEmpty      :: Stack a -> Bool
isEmpty       = null

instance Eq a => Eq (Stack a) where
    s1 == s2 | isEmpty s1 = isEmpty s2
             | isEmpty s2 = isEmpty s1
             | otherwise  = topOf s1 == topOf s2 &&
                 pop s1 == pop s2
```

In permutations.hs taken from here:

```
--selections :: [a] -> [(a, [a])]
selections [] = []
selections (x:xs) =
   (x, xs) :
   [ (z, x:zs) | (z, zs) <- selections(xs) ]

permutations [] = [[]]
permutations xs =
   [ y:zs |
    (y, ys) <- selections xs,
    zs <- permutations ys ]
```

Permutations Log

```
Prelude> :l "permutations.hs"
...
Main> permutations []
[[]]
(12 reductions, 25 cells)
Main> permutations [1, 2, 3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
(184 reductions, 471 cells)
Main> permutations [1]
[[1]]
(24 reductions, 47 cells)
Main>
```

Edit Distance

From Thompson:

```
data Edit
  = Change Char
  | Copy
  | Delete
  | Insert Char
  | Kill
  deriving (Eq, Show)
```

Edit Distance Continued

```
transform :: String -> String -> [Edit]

transform [] [] = []
transform string [] = [Kill]
transform [] string = map Insert string
transform (a:x) (b:y)
  | a == b =     Copy : transform x y
  | otherwise = best [
     Delete : transform x (b:y),
     Insert b : transform (a:x) y,
     Change b : transform x y ]
```

Edit Distance Continued

```
best :: [[Edit]] -> [Edit]
best [a] = a
best (a:x)
   | cost a <= cost b = a
   | otherwise        = b
     where b = best x

cost :: [Edit] -> Int
cost = length . filter (/=Copy)
```

Edit Distance Log

```
Main> transform "abc1" "ac2"
[Edit_Copy,Edit_Delete,Edit_Copy,Edit_Change '2']
(1045 reductions, 1422 cells)
Main> transform "abcd" "1a2"
[Edit_Insert '1',Edit_Copy,Edit_Insert
 '2',Edit_Kill]
(7122 reductions, 9088 cells)
Main>
```

Currying

In simple.hs

```
simple a b c = a * (b + c)

Hugs> :l "programs/simple.hs"
Main> :t simple
simple :: Num a => a -> a -> a -> a
Main> :t (simple 5)
simple 5 :: Num a => a -> a -> a
Main> :t (simple 5 3)
simple 5 3 :: Num a => a -> a
Main> :t (simple 5 3 2)
simple 5 3 2 :: Num a => a
Main> (((simple 5) 3) 2)
25
Main>
```

Currying Simplification

In listsumprod.hs

```
listSum, listProd :: [Float] -> Float
listSum xs = foldl (+) 0 xs
listProd xs = foldl (*) 1 xs
```

In listsumprod-curry.hs

```
listSum, listProd :: [Float] -> Float
listSum = foldl (+) 0
listProd = foldl (*) 1
```

Curying Infix Operators using Sections

- `(x / y)` is equivalent to a function `f1 x y = x / y`.
- `(x /)` is equivalent to a function `f1 y = x / y`.
- `(/ y)` is equivalent to a function `f1 x = x / y`.

```
Main> (10 / 5)
2.0
Main> (10 /) 5
2.0
Main> (/ 5) 10
2.0
Main> (/) 10 5
2.0
Main>
```

Another Sections Example

posints.hs

```
posInts1  :: [Integer] -> [Bool]
posInts1 xs = map test xs
              where test x = x > 0

posInts2  :: [Integer] -> [Bool]
posInts2 xs = map (> 0) xs

posInts3  :: [Integer] -> [Bool]
posInts3 = map (> 0)
```

reverse.hs

```
rev1 :: [a] -> [a]
rev1 [] = []
rev1 (x:xs) = rev1 xs ++ [x]

rev2 :: [a] -> [a]
rev2 xs = rev2Aux [] xs
   where rev2Aux acc [] = acc
         rev2Aux acc (x:xs) = rev2Aux (x:acc) xs
```

```
rev3:: [a] -> [a]
rev3 xs = foldl revOp [] xs
               where revOp acc x = x:acc

-- Using flip f x y = f y x defined in Prelude.
-- revOp acc x = flip (:) acc x
-- 2 applications of currying simplification:
revOp = flip (:)
rev4 :: [a] -> [a]
rev4 = foldl (flip (:)) []
```

- We could type +

```
(+) :: Integer -> Integer -> Integer
```

  but that would not allow us to add floats, or complex numbers.
- We would have separate addition functions `addInteger`, `addFloat`, `addComplex`, but that would not be satisfactory.

Qualified Types

- Giving `(+)` the polymorphic type `a -> a -> a` would be too general, because the type-variable `a` is implicitly universally quantified.
- Solution is to use a *qualified type*:

```haskell
(+) :: Num a => a -> a -> a
```

  which is read as "for all types `a` that are members of the class `Num`, `(+)` has type `a -> a -> a`".

Equality

- Equality cannot be computed for all types; for example, one cannot determine the equality of two infinite lists or two functions. Hence *computational equality* is weaker than *full equality*.
- So we can compute equality for some types but not for others. We say that a type implements equality if it is a member of type type class `Eq` which defines the function: `(==) :: Eq a => a -> a -> Bool`.
- `Integer` and `Char` are instances of `Eq`. `42 == 43` and `'a' == 'a'` are well-typed but `42 == 'a'` is not.

Equality Continued

- Type constraints can be propagated through polymorphic data types. Hence `[10, 12] == [10, 12]` and `['a', 'b'] == "ac"` are well-typed. On the other hand, `[10, 12] == "ac"` is not.
- Qualified types also propagate through function definitions. Consider `member` function:

```
member x [] = False
member x (y:ys) = (x == y) || member x ys
```

has type `Eq a => a -> [a] -> Bool`.

- In Haskell's *Standard Prelude*:

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

  says "a type `a` is an instance of the class `Eq` iff there is an operation `(==) :: a -> a -> Bool` defined on it."
- We can say that a particular type are instances of `Eq`:

  ```
  instance Eq Integer where
     x == y = IntegerEq x y
  ```

Equality over User Data Types

Given

```
data Tree a =
  Leaf a
  | Node (Tree a) (Tree a)
```

define equality as:

```
instance Eq a => Eq (Tree a) where
Leaf a == Leaf b = a == b
Node t1 t2 == Node s1 s2 = t1 == s1 && t2 == s2
_ == _ = False
```

Full Definition of Equality

Definition of `Eq` in Haskell's *Standard Prelude*:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

defines 2 operations with *default methods* for each operator.

Polymorphism versus Type Classes

- *Polymorphism* captures similar structure over different values. For example, a sequence of integers, sequence of strings, etc. can be captured by a polymorphic *List*.
- *Type classes* capture similar operations over different structures. For example, equality of integers, equality of trees, etc. can be captured by a class `Eq`.

Type Class Inheritance

`Ord` inherits all the operations in `Eq`:

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

`Eq` is a super-class of `Ord`.
Note the previous definition of `quicksort` had type:

```
Ord a => [a] -> [a]
```

Standard Prelude Type Classes

Some useful type classes contained in the Standard Prelude:

- `Eq` for `==`, `/=`.
- `Ord` for `<`, `<=`, `>`, `>=`, `max`, `min`, `compare`, `Ordering`
- `Show` for converting types to character strings.
- `Read` for converting character strings to types.
- `Num` for numeric types.

hello.hs

```
hello =
  do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Hello " ++ name ++ "!"
```

```
Main> hello
Hello, what's your name?
Tim
Hello Tim!

Main> :t hello
hello :: IO ()
Main> :t putStrLn
putStrLn :: String -> IO ()
Main> :t getLine
getLine :: IO String
Main>
```

I/O Actions

- If a return value has type involving `IO`, then it means that function has a side-effect.
- All I/O actions have the type `IO t`.
- Produces side-effect when *performed*, not *evaluated*.
- I/O is performed within another I/O action or at the top-level.

I/O Actions Continued

- `do` allows *sequencing* I/O actions.
- `<-` is an operator which extracts a value of type `t` from `IO t`. Hence `getLine :: IO String`.

Restrict side-effects. `nameToGreet` below is *pure* and has type `String -> String`.

hello2.hs

```
nameToGreet name =
  "Hello " ++ name ++ "!"

hello2 =
  do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ nameToGreet name
```

Lazy I/O

- Function `hGetContents :: Handle -> IO String` returns all the contents from current position in Handle.
- Does not produce any I/O when called.
- I/O is performed lazily as return value is processed.
- Can conceptually have all contents of a 4GB file in memory. The contents can then be processed using pure functions with I/O occurring lazily behind the scenes.

References

Hal Daume III, Yet Another Haskell Tutorial.
haskell.org. Particularly see the documentation.
Paul Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.
Bryan O'Sullivan, John Goerzen and Don Stewart, *Real World Haskell*, O'Reilly, 2009.
School of Haskell
Simon Thompson, *Haskell: The Craft of Functional Programming*, Second Edition, Addison-Wesley, 1999.