1. Referential transparency, no side effects
   “substitution of equals for equals”
2. Function definitions can be used
   Suppose \( f \) is defined to be the function \((\text{fn } x=>\text{exp}), \text{then } f \ (\text{arg}) \) can be replaced by \( \text{exp}[x := \text{arg}] \)
3. Lists not arrays
4. Recursion not iteration
5. Universal parameteric polymorphism, type reconstruction
6. Higher-order functions
   New idioms, total procedural abstraction
fun test (x) = if x>20 then "big" else "small"

test (sos (3,4))
==> test(25)
==> if 25>20 then "big" else "small"
==> "big"
fun square x = x * x;
fun sos (x,y) = (square x) + (square y);

sos (3,4)
==> (square 3) + (square 4) [Def’n of sos]
==> 3*3 + (square 4) [Def’n of square]
==> 9 + (square 4) [Def’n of *]
==> 9 + 4*4 [Def’n of square]
==> 9 + 16 [Def’n of *]
==> 25 [Def’n of +]
History of Functional Languages

1959  LISP: List processing, John McCarthy
1975  Scheme: MIT
1977  FP: John Backus
1980  Hope: Burstall, McQueen, Sannella
1984  COMMON LISP: Guy Steele
1985  ML: meta-language (of LCF), Robin Milner
1986  Miranda: Turner
1990  Haskell: Hudak & Wadler editors
xkcd—a webcomic of romance, sarcasm, math, and language by Randall Munroe
Lazy: don’t evaluate the function (constructor) arguments until needed (call-by-name), e.g., Haskell. Permits infinite data structures.

Eager: call-by-value, e.g., ML
ML and Haskell

- Similar to ML: functional, strongly-typed, algebraic data types, type inferencing
- Differences: no references, exception handling, or side effects of any kind; lazy evaluation, list comprehensions
Functions

\[ x \rightarrow 2 \times x \]
Cons 1 (Cons 2 Nil)
1: 2: Nil
[1, 2]
[1, 2, 3, 4]
Efficiency not determined by the letters in the name.
A function defined on a type can only be called by constructing the type (with all its elements) and then apply the functions. A curried function is more flexible. Apply the function to each argument individually is possible and many times the result are useful.

\[
\begin{align*}
    \text{add3} \ (x, y, z) &= x+y+z+1 \\
    \text{add3}' \ x \ y \ z &= x+y+z+1 \\
    \text{add3} \ (1, 2, 3) \\
    \text{add3} \ 1 \ 2 \ 3 \\
    \text{add3} \ 1 \ 2 \\
    \text{add3} \ 1 \\
\end{align*}
\]
Higher-Order Functions

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow x+y+4 \]
\[ f_2 = \lambda x \ y \rightarrow x+y+4 \]
\[ f_3 \ x = \lambda y \rightarrow x + y + 4 \]
\[ f_4 \ x \ y = x + y + 4 \]

-- pleasing and familiar symmetry of definition and use

\[ f_4 \ 3 \ 4 \]
Higher-Order Functions

\[ \text{con1} = \lambda x \rightarrow \lambda y \rightarrow x \]
\[ \text{con2} = \lambda x y \rightarrow x \]
\[ \text{con3} x = \lambda y . x \]
\[ \text{con4} x y = x \]

\[ \text{app1} = \lambda f \rightarrow \lambda x \rightarrow f x \]
\[ \text{app2} = \lambda f x \rightarrow f x \]
\[ \text{app3} f = \lambda x \rightarrow f x \]
\[ \text{app4} f x = f x \]
Partial Application

Any curried function may be called with fewer arguments than it was defined for. The result is a *function* of the remaining arguments.

If $f$ is a function $\text{Int} \rightarrow \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool}$, then

\begin{align*}
f & : \text{Int} \rightarrow \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool} \\
f\ 2 & : \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool} \\
f\ 2\ True & : \text{Int} \rightarrow \text{Bool} \\
f\ 2\ True\ 3 & : \text{Bool} \\
\end{align*}

Higher-order functions after lists.
Curry

\[ A \times B \rightarrow C \cong A \rightarrow B \rightarrow C \]

curry \quad f = \lambda x \rightarrow \lambda y \rightarrow f(x, y)

uncurry \quad g = \lambda (x, y) \rightarrow g x y
\texttt{curry} = \lambda f \rightarrow \lambda x \rightarrow \lambda y \rightarrow f(x,y)

\texttt{curry} :: \lambda f \rightarrow \lambda x \rightarrow \lambda y \rightarrow f(x,y) :: A

\texttt{curry} :: (f::B) \rightarrow (\lambda x \rightarrow \lambda y \rightarrow f(x,y)) :: C

\texttt{curry} :: (f::B) \rightarrow (x::D) \rightarrow (\lambda y \rightarrow f(x,y)) :: E

\texttt{curry} :: (f::B) \rightarrow (x::D) \rightarrow (y::F) \rightarrow (f(x,y)) :: G

\texttt{curry} :: (f::H\rightarrow G) \rightarrow (x::D) \rightarrow (y::F) \rightarrow (f(x,y)) :: G

\texttt{curry} :: (f::(D,F) \rightarrow G) \rightarrow (x::D) \rightarrow (y::F) \rightarrow (f(x,y)) :: G

\texttt{curry} :: ((a,b)\rightarrow c) \rightarrow a \rightarrow b \rightarrow c
map pattern
filter pattern
fold right pattern
Haskell Fold

\[
\begin{align*}
\text{foldr} & \quad :: \ (b \to a \to a) \to a \to [b] \to a \\
\text{foldr}\ f\ z\ [] & = z \\
\text{foldr}\ f\ z\ (x:xs) & = f\ x\ (\text{foldr}\ f\ z\ xs)
\end{align*}
\]

\[
\begin{align*}
\text{foldl} & \quad :: \ (a \to b \to a) \to a \to [b] \to a \\
\text{foldl}\ f\ z\ [] & = z \\
\text{foldl}\ f\ z\ (x:xs) & = \text{foldl}\ f\ (f\ z\ x)\ xs
\end{align*}
\]

\[
\begin{align*}
\text{foldl}' & \quad :: \ (a \to b \to a) \to a \to [b] \to a \\
\text{foldl}'\ f\ z0\ xs & = \text{foldr}\ f'\ \text{id}\ xs\ z0 \\
\text{where} & \quad f'\ x\ k\ z = k\ (!!\ f\ z\ x
\end{align*}
\]

[Real World Haskell says never use foldl instead use foldl’.]
Evaluates its first argument to head normal form, and then returns its second argument as the result.

\[
\text{seq} :: a \to b \to b
\]

Strict (call-by-value) application, defined in terms of `seq`.

\[
(\text{$!$}) :: (a \to b) \to a \to b
\]

\[
f \text{$!$} x = x \ `\text{seq}` \ f x
\]
“A tutorial on the universality and expressiveness of fold” by Graham Hutton.
Fold

\[
\text{foldr} \otimes z[x_1, x_2, \ldots, x_n] = x_1 \otimes (x_2 \otimes (\ldots (x_n \otimes z) \ldots))
\]

\[
\text{foldr } f \ z \ [x_1, x_2, \ldots, x_n] = x_1 \ f \ (x_2 \ f \ (\ldots (x_n \ f \ z) \ldots))
\]
Fold

\[ \text{foldl}(\otimes) z[x_1, x_2, \ldots, x_n] = \ldots(\ldots(\ldots(\ldots z \otimes x_1 \otimes x_2) \ldots) \otimes x_n \ldots) \text{ f } x_n \]
One important thing to note in the presence of lazy, or normal-order evaluation, is that foldr will immediately return the application of f to the recursive case of folding over the rest of the list. Thus, if f is able to produce some part of its result without reference to the recursive case, and the rest of the result is never demanded, then the recursion will stop. This allows right folds to operate on infinite lists. By contrast, foldl will immediately call itself with new parameters until it reaches the end of the list. This tail recursion can be efficiently compiled as a loop, but can’t deal with infinite lists at all – it will recurse forever in an infinite loop.
Another technical point to be aware of in the case of left folds in a normal-order evaluation language is that the new initial parameter is not being evaluated before the recursive call is made. This can lead to stack overflows when one reaches the end of the list and tries to evaluate the resulting gigantic expression. For this reason, such languages often provide a stricter variant of left folding which forces the evaluation of the initial parameter before making the recursive call, in Haskell, this is the foldl’ (note the apostrophe) function in the Data.List library. Combined with the speed of tail recursion, such folds are very efficient when lazy evaluation of the final result is impossible or undesirable.
Haskell Fold

```
sum' = foldl (+) 0
product' = foldl (*) 1
and' = foldl (&&) True
or' = foldl (||) False
concat' = foldl (++) []
composel = foldl (.) id
composer = foldr (.) id
length = foldl (const (+1)) 0
list_identity = foldr (:) []
reverse' = foldl (flip (:)) []
unions = foldl Set.union Set.empty
elem' a = foldr (\x r -> (compare a x == EQ) || r) False
append = foldr (\x l -> x:l)
```
Haskell Fold

```haskell
reverse = foldl (\ xs x -> xs ++ [x]) []
map f = foldl (\ xs x -> f x : xs) []
filter p = foldl (\ xs x -> if p x then x:xs else xs) []
```
Haskell Fold

If this is your pattern

\[
\begin{align*}
g \, [] & = v \\
g \, (x:x) & = f \, x \, (g \, x)
\end{align*}
\]

then

\[
g = \text{foldr} \ f \ v
\]
Haskell Data Structures

```haskell
data Bool = False | True

data Color = Red | Green | Blue
  deriving Show

data Day = Mon | Tue |Wed |Thu |Fri |Sat |Sun
  deriving (Show, Eq, Ord)
```

Types and constructors capitalized.
Show allows Haskell to print data structures.
Constructors can take arguments.

```haskell
data Shape = Circle Float | Rectangle Float Float

deriving Show

area (Circle r) = pi*r*r
area (Rectangle s1 s2) = s1*s2
```
Haskell Types

Type constructors can take types as parameters.

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

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing = n
maybe _ f (Just x) = f x

data Either a b = Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x) = f x
either _ g (Right y) = g y
```
Data types can be recursive, as in lists:

```haskell
data Nat = Nil | Succ Nat
data IList = Nil | Cons Integer IList
data PolyList a = Nil | Cons a (PolyList a)
```
Haskell Trees

See Hudak PPT, Ch7.

```haskell
data SimpleTree = SimLeaf | SimBranch SimpleTree SimpleTree

data IntegerTree = IntLeaf Integer | IntBranch IntegerTree IntegerTree

data InternalTree a = ILeaf | IBranch a (InternalTree a) (InternalTree a)

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

data FancyTree a b = FLeaf a | FBranch b (FancyTree a b) (FancyTree a b)

data GTree = GTree [GTree]

data GPTree a = GPTree a [GPTree a]
```
Nested Types

```haskell
data List a = NilL | ConsL a (List a)
data Nest a = NilN | ConsN a (Nest (a,a))
data Bush a = NilB | ConsB a (Bush (Bush a))

data Node a = Node2 a a | Node3 a a a
data Tree a = Leaf a | Succ (Tree (Node a))
```

Hinze, Finger Trees.
Haskell Classes

```haskell
next :: (Enum a, Bounded a, Eq a) => a -> a
next x | x == maxBound = minBound
        | otherwise = succ x
```
Haskell Classes

data Triangle = Triangle

data Square = Square

data Octagon = Octagon

class Shape s where
  sides :: s -> Integer

instance Shape Triangle where
  sides _ = 3

instance Shape Square where
  sides _ = 4

instance Shape Octagon where
  sides _ = 8
Haskell

input stream --> program --> output stream
Real World

[Char] --> program --> [Char]
Haskell World

```
module Main where

main = do
    input <- getContents
    putStrLn $ unlines $ f $ lines input

countWords :: String -> String
countWords = unlines . format . count . words

count :: [String] -> [(String, Int)]
count = map (\ws -> (head ws, length ws)) . groupBy (==) . sort
```
Haskell

input stream --> program --> output stream
Real World

[Char] --> program --> [Char]
Haskell World

```haskell
module Main where

main = interact countWords

countWords :: String -> String
countWords = unlines . format . count . words

count :: [String] -> [(String, Int)]
count = map (\ws->(head ws, length ws)) . groupBy (==) . sort
```