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 $(\text{fn } x \Rightarrow \text{exp})$, then $f(\text{arg})$ can be replaced by $\text{exp}[x := \text{arg}]$
3. Lists not arrays
4. Recursion not iteration
5. Higher-order functions
   New idioms, total procedural abstraction
Rewriting

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 +]

Language of expressions only, no statements.

fun test (x) = if x>20 then "big" else "small"

test (sos (3,4))
===> test(25)
===> if 25>20 then "big" else "small"
Canonical value. A canonical value is one which cannot be rewritten further. For example, $2 + 3$ is not canonical, it evaluates to 5; 5 is a canonical value. See *canonical* in the “The on-line hacker Jargon File,” version 4.4.7, 29 Dec 2003.
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
LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.

HUH?

SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.

AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.

MY GOD

IT'S FULL OF CAR'S

THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.

TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.

TRULY, THIS WAS

THE LANGUAGE

FROM WHICH THE

GODS WROUGHT

THE UNIVERSE.

NO, IT'S NOT.

IT'S NOT?

I MEAN, OUSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.
Functional Languages

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
Introduction to Haskell

1. Haskell (1.0) 1990
2. By 1997 four iterations of language design (1.4)
Salient Features of Haskell

1. Strongly-typed, lazy, functional language
2. Polymorphic types, type inference
3. Algebraic type definitions
4. Pattern matching function definitions
5. System of classes
6. Interactive
GHC Interactive, version 6.4.1, for Haskell 98.
http://www.haskell.org/ghc/
Type :? for help.

Loading package base-1.0 ... linking ... done.
Prelude> :load u:main
Compiling Main
Ok, modules loaded: Main.
*Main> f "abcdefgh"
"hgfedcba"
*Main> :reload
Ok, modules loaded: Main.
*Main> :quit

```
emacs@CS245-50586:~$ module Main where

import System.IO as IO

-- main program to reverse each line of input
main =
do IO.getContents >>= IO.putStrLn . unlines . map f . lines

f line = reverse line

-- built-in functions:
-- unlines :: [String] -> String ; breaks input into separate lines
-- lines :: String -> [String] ; combines separate lines into one string
-- reverse :: [a] -> [a] ; reverse list back to front
```
Information about Haskell

O’Sullivan, Goerzen, Stewart *Real World Haskell*

Hutton, Graham, *Programming in Haskell.*
O’Donnell et al., *Discrete Mathematics Using a Computer.*
Introduction to SML

1. Robin Milner, Turing Award winner 1991
2. Metalanguage (ML) in Logic of Computable Functions (LCF) 1980s
3. Actually general purpose language
5. AT&T (D. McQueen), Princeton (A. Appel) implementation
Salient Features of SML

1. Strongly-typed, eager, functional language
2. Polymorphic types, type inference
3. Algebraic type definitions
4. Pattern matching function definitions
5. Exception handling
6. Module (signatures/structures) system
7. Interactive
Information about ML


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

```haskell
fac n = if n==0 then 1 else n * fac (n-1)
data Tree = Leaf | Node (Tree, String, Tree)
size (Leaf) = 1
size (Node (l,_,r)) = size (l) + size (r)
squares = [ n*n | n <- [0..] ]
pascal = iterate (\row->zipWith (+) ([0]++row) (row ++[0])) ([1])
```
Haskell List Comprehension

\[ e \mid x_1 \leftarrow l_1, \ldots, x_m \leftarrow l_m, P_1, \ldots, P_n \]

\( e \) is an expression, \( x_i \) is a variable, \( l_i \) is a list, \( P_i \) is a predicate

\[ x^2 \mid x \leftarrow [1..10], \text{even } x \]
\[ x^2 \mid x \leftarrow [2,4..10] \]
\[ x+y \mid x \leftarrow [1..3], y \leftarrow [1..4] \]

\( \text{perms} \) \[ [] = [[]] \]
\( \text{perms} \) \( x \) = \[ a:y \mid a \leftarrow x, y \leftarrow \text{perms} \ (x \setminus [a]) \]

\( \text{quicksort} \) \[ [] = [] \]
\( \text{quicksort} \) \( s : xs \) =
\( \text{quicksort} \ [\ x \mid x \leftarrow xs, x < s \] ++ \ [s] ++ \text{quicksort} \ [\ x \mid x \leftarrow xs, x > s \] \)
Patterns are a very natural way of expression complex problems. Consider the code to re-balance red-black trees. This is usually quite complex to express in a programming language. But with patterns it can be more concise. Notice that constructors of user-defined types (line RBTREE) as well as pre-defined types (like list) can be used in patterns.
data Color = R | B deriving (Show, Read)
data RBTree a = Empty | T Color (RBTree a) a (RBTree a) deriving (Show, Read)

balance :: RBTree a -> a -> RBTree a -> RBTree a
balance (T R a x b) y (T R c z d) = T R (T B a x b) 
balance (T R (T R a x b) y c) z d = T R (T B a x b) 
balance (T R a x (T R b y c)) z d = T R (T B a x b) 
balance a x (T R b y (T R c z d)) = T R (T B a x b) 
balance a x (T R (T R b y c) z d) = T R (T B a x b) 
balance a x b = T B a x b
Functions

Prelude> : m Text.Show.Functions
Prelude Text.Show.Functions> \x->x+1
 <function>
Prelude Text.Show.Functions> :t \x->x+(1::Int)
 \x->x+(1::Int) :: Int -> Int
Prelude Text.Show.Functions> :t \x->x+1
 \x->x+1 :: (Num a) => a -> a
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 $\text{Int} \to \text{Bool} \to \text{Int} \to \text{Bool}$, then

\[
\begin{align*}
f & \quad :: \ \text{Int} \to \text{Bool} \to \text{Int} \to \text{Bool} \\
f \ 2 & \quad :: \ \text{Bool} \to \text{Int} \to \text{Bool} \\
f \ 2 \ \text{True} & \quad :: \ \text{Int} \to \text{Bool} \\
f \ 2 \ \text{True} \ 3 & \quad :: \ \text{Bool}
\end{align*}
\]

Higher-order functions after lists.
Cons 1 (Cons 2 Nil)
1 : 2 : Nil
[1, 2]
“A tutorial on the universality and expressiveness of fold” by Graham Hutton.
Fold

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

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

\[
\text{foldl } \bigotimes z\{x_1, x_2, \ldots, x_n\} = (\ldots((z \bigotimes x_1) \bigotimes x_2)\ldots) \bigotimes x_n
\]

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

foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z0 xs = foldr f' id xs z0
  where f' x k z = k $! f z x

[Real World Haskell says never use foldl instead use foldl'.]
Haskell Fold

Evaluates its first argument to head normal form, and then returns its second argument as the result.

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

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

\[ (\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b \]
\[ f \$! x = x \ '\text{seq}' \ f \ x \]
Haskell Fold

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

```haskell
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
```
Haskell Fold

\[
\begin{align*}
\text{reverse} & = \text{foldl} \ (\ \lambda \ xs \ x \to \ xs \ ++ \ [x]) \ [] \\
\text{map \ f} & = \text{foldl} \ (\ \lambda \ xs \ x \to \ f \ x \ : \ xs) \ [] \\
\text{filter \ p} & = \text{foldl} \ (\ \lambda \ xs \ x \to \text{if} \ p \ x \ \text{then} \ x:xs \ \text{else} \ xs) \ []
\end{align*}
\]
If this is your pattern

\[
g \ [ ] = v \\
g \ (x:xs) = f \ x \ (g \ xs)
\]

then

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

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.
Haskell 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 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 Types

Type constructors can type 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
```
Haskell Lists

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
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

\begin{verbatim}
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))
\end{verbatim}

Hinze, Finger Trees.
Haskell

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

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

```haskell
module Main where

main = do
  input <- getContents
  putStr $ 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

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