Introduction to Functional Languages

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 \( (\text{fn } x=>\text{exp}) \), then \( f (\text{arg}) \) can be replaced by \( \text{exp}[x := \text{arg}] \)

3. Lists not arrays

4. Recursion not iteration

5. Universal parametric polymorphism, type reconstruction

6. Higher-order functions
   New idioms, total procedural abstraction
The power of fun programming derives from:

- constant meaning (referential transparency)
- flexibility of high-order functions
- goal direction (no storage management)
• In a functional language an expression is the program plus its input.
• Expressions have parts which can be reduced $\triangle$

\[ \triangle \rightarrow \blacktriangle \]

• Reduction continues until no more reducible parts exist
• The result corresponds to the output.
Schematic Representation of Reduction

\[ \triangle \quad \triangle \triangle \]

\[ \blacktriangle \quad \triangle \triangle \]

\[ \blacktriangle \quad \blacktriangle \quad \triangle \]

\[ \blacktriangle \quad \blacktriangle \quad \blacktriangle \]
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"
===> "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
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisp</td>
<td>LISP 1.5, LISP 2 (abandoned)</td>
<td>Maclisp</td>
<td>Interlisp</td>
<td>Lisp Machine Lisp</td>
<td>Scheme</td>
<td>R5RS</td>
<td>R6RS</td>
<td>R7RS small</td>
<td>Common Lisp</td>
<td>Le Lisp</td>
<td>T</td>
<td>Emacs Lisp</td>
<td>AutoLISP</td>
<td>OpenLisp</td>
</tr>
</tbody>
</table>
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
ITS 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.

IT'S NOT?
I MEAN, OSTEINSLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

No, it's not.
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
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


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

\( x \rightarrow 2 \times x \)
Efficiency not determined by the letters in the name.
Functions

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.

\[
\text{add3 (x,y,z) = x+y+z+1} \\
\text{add3 \ ' x y z = x+y+z+1}
\]

\[
\text{add3 (1,2,3)} \\
\text{add3 1 2 3} \\
\text{add3 1 2} \\
\text{add3 1}
\]
Higher-Order Functions

A function is called *higher-order* if it takes a function as an argument or returns a function as a result.

```haskell
let twice f x = f (f x);;
let twice = fun f x -> f (f x);;
let twice = fun f -> fun x -> f (f x);;
```

twice is higher-order because it takes a function as an argument.
Higher-order functions are useful

- “by allowing common programming patterns to be encapsulated as functions” [Hutton, page 74]
- “to define domain-specific languages within Haskell” [Hutton, page 74]
- properties of function can be used in proof of correctness
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 $\text{Int} \to \text{Bool} \to \text{Int} \to \text{Bool}$, then

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

1 Common Patterns

2 Function Spaces, Currying

3 Haskell Data Structures
Two of the most common list patterns are higher-order.

1. map
2. fold (aka reduce)

Also filter.
map pattern
filter pattern
fold left pattern
fold right pattern
Haskell Fold

\[
\text{foldr} :: (b \to a \to a) \to a \to [b] \to a
\]
\[
\text{foldr } f \; z \; [] = z
\]
\[
\text{foldr } f \; z \; (x:xs) = f \; x \; (\text{foldr } f \; z \; xs)
\]

\[
\text{foldl} :: (a \to b \to a) \to a \to [b] \to a
\]
\[
\text{foldl } f \; z \; [] = z
\]
\[
\text{foldl } f \; z \; (x:xs) = \text{foldl } f \; (f \; z \; x) \; xs
\]

\[
\text{foldl}' :: (a \to b \to a) \to a \to [b] \to a
\]
\[
\text{foldl}' \; f \; z0 \; xs = \text{foldr } f' \; \text{id} \; xs \; z0
\]
\[
\text{where } f' \; x \; k \; z = k \; \_! \; f \; z \; x
\]

\text{foldl} \text{ is tail recursive!}

[\textit{Real World Haskell} says never use foldl instead use foldl'.]
Haskell Fold

\[
foldr :: (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a
\]

\[
foldr f z [] = z
\]

\[
foldr f z (x:xs) = f x (foldr f z xs)
\]

\[
foldk k f z [] = k z
\]

\[
foldk k f z (x:xs) = foldk (k.(\ v \rightarrow f x v)) f z xs
\]

\[
foldr = foldk id
\]
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

“A tutorial on the universality and expressiveness of fold” by Graham Hutton (J. Fun Prog, 1999)

Wikipedia: Fold (higher order function)
> foldr (\x y -> concat ["(" ,x ,"+" ,y ,")"] ) "0" (map show [1..13])
"((1+(2+(3+(4+(5+(6+(7+(8+(9+(10+(11+(12+(13+0))))))))))))))")

> foldl (\x y -> concat ["(" ,x ,"+" ,y ,")"] ) "0" (map show [1..13])
"(((((((((((((0+1)+2)+3)+4)+5)+6)+7)+8)+9)+10)+11)+12)+13)"
Fold

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

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

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

\[
\text{fold} \left( z_1, x_2, \ldots, x_n \right) = \left( \ldots \left( z_1 \otimes x_1 \otimes x_2 \right) \ldots \right) \otimes x_n
\]

\[
\text{fold} f z [x_1, x_2, \ldots, x_n] = (\ldots((z \ 'f' \ x_1) \ 'f' \ x_2) \ldots) \ 'f' \ xn
\]

\[
\text{fold} f z [x_1, x_2, \ldots, x_n] = f (\ldots(f (f z x_1) x_2) \ldots) \ xn
\]
One important thing to note in the presence of lazy, or normal-order evaluation, is that \( \text{foldr} \) 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, \( \text{foldl} \) 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 (Associative Operations)

sumR = foldr (+) 0
sumL = foldl (+) 0
productR = foldl (*) 1
productL = foldl (*) 1
andR = foldl (&&) True
andL = foldl (&&) True
orR = foldr (||) False
orL = foldl (||) False
concatR = foldl (++) []
concatL = foldl (++) []
unionsR = foldl Set.union Set.empty
unionsL = foldl Set.union Set.empty
composeR = foldr (.) id
composeL = foldl (.) id
Haskell Fold

```haskell
lengthR = foldr (const (+1)) 0
lengthL = foldl (const . (+1)) 0

idR = foldr (:) []
idL = foldl (\xs x -> xs++[x]) [] = foldl snoc []

appendR = foldr (:) = foldr (\y l -> y:l)
appendL xs = foldl {- impossible -} {- impossible -}

reverseR = foldr (((flip (++)).(::[]))) = foldr (\x xs -> xs ++ [x])
reverseL = foldl (flip ()): [] = foldl (\xs x -> x : xs) []

elemR a = foldr (\x r -> (compare a x == EQ) || r) False
elemL a = foldl (\r x -> (compare a x == EQ) || r) False
```
Haskell Length Using Fold Left

```haskell
lengthL ["ab", "", "def"]
foldl (const . (+1)) 0 ["ab", "", "def"]
(const .(+1)) ((const .(+1)) ((const .(+1)) 0 "ab") ")") "def"
(const .(+1)) ((const .(+1)) ((const 1) "ab") ")") "def"
(const .(+1)) ((const .(+1)) 1 ")") "def"
(const .(+1)) (const 2 ")") "def"
(const .(+1)) 2 "def"
const 3 "def"
3
```
Haskell Length Using Fold Right

```
lengthR ["ab", ",", "def"]
foldr (const (+1)) 0 ["ab", ",", "def"]
(const (+1)) "ab" ((const (+1)) "," (const (+1) "def" 0))
(+1) ((const (+1)) "," (const (+1) "def" 0))
(+1) ((+1) (const (+1) "def" 0))
(+1) ((+1) ((+1) 0))
(+1) ((+1) 1)
(+1) 2
3
```
Haskell Reverse Using Fold Left

reverseL [1,3,5,7]
foldl (flip (:)) [] [1,3,5,7]

flip(:) ((flip(:)) ((flip(:)) [1] 3) 5) 7
flip(:) ((flip(:)) [3,1] 5) 7
flip(:) [5,3,1] 7
[7,5,3,1]
Haskell Reverse Using Fold Right

reverseR [1,3,5,7]
foldR (f) [] [1,3,5,7]
(f) 1 ((f) 3 ((f) 5 ((f) 7 [])))
(++)[1] ((++)[3] ((++)[5] [7]))
(++)[1] ((++)[3] [7,5])
(++)[1] [7,5,3]
[7,5,3,1]
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) []
```
If this is your pattern

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

then

g = foldr f v
Haskell Fold

Left fold is tail recursive. Can one make right fold tail recursive? Solve the equation

$$\text{foldr} = \text{foldTR \ id}$$

We need

$$\text{foldTR \ k \ f \ z \ [\text{]} = \{- \ldots -\}}$$
$$\text{foldTR \ k \ f \ z \ (x:xs) = \text{foldTR \ k' \ f \ z \ xs}}$$

where

$$k' \ v = \{- \ldots -\}}$$
Haskell Fold

Left fold is tail recursive. Can one make right fold tail recursive? Solve the equation

\[
\text{foldr} = \text{foldTR} \ id
\]

We need

\[
\text{foldTR} \ k \ f \ z \ [] = \{- \ . \ . \ -\}
\text{foldTR} \ k \ f \ z \ (x:xs) = \text{foldTR} \ k' \ f \ z \ xs
\text{where}
\begin{align*}
k' \ v &= \{- \ . \ . \ -\} \\
k' \ v &= k (f \ x \ v)
\end{align*}
\]
Outline

1. Common Patterns
2. Function Spaces, Currying
3. Haskell Data Structures
Examine currying first; before higher-order, recursive list patterns (map, filter, foldl and foldr).

Every function take one argument. Every function $f : D \rightarrow B$ has one domain and one range.
Currying

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

The function space \( A \times B \rightarrow C \) is isomorphic to the function space \( A \rightarrow B \rightarrow C \).

\[
\text{curry } \ f \ x \ y \ = \ f \ (x,y) \\
\text{uncurry } \ g \ (x,y) \ = \ g \ x \ y
\]
Currying

These functions can be written more explicitly without the special Haskell function declaration notation.

\[
\text{curry } f \, x \, y = f \,(x, y) \\
\text{uncurry } g \,(x, y) = g \, x \, y
\]

\[
\text{curry } f = \lambda \, x \to \lambda \, y \to f \,(x, y) \\
\text{uncurry } g = \lambda \,(x, y) \to g \, x \, y
\]

\[
\text{curry } = \lambda f \to \lambda \, x \to \lambda \, y \to f \,(x, y) \\
\text{uncurry } = \lambda g \to \lambda \,(x, y) \to g \, x \, y
\]
A careful derivation of the types of functions follows.

\[
\text{curry} :: ((a,b)\rightarrow c) \rightarrow a \rightarrow b \rightarrow c
\]
\[
\text{curry} = \lambda f \rightarrow \lambda x \rightarrow \lambda y \rightarrow f(x,y)
\]

\[
\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a,b)\rightarrow c
\]
\[
\text{uncurry} = \lambda g \rightarrow \lambda (x,y) \rightarrow g x y
\]
curry :: ((a,b)->c) -> a -> b -> c
curry = \f -> \ x -> \ y -> f (x,y)

curry :: [ \f -> \ x -> \ y -> f (x,y) ] :: A
-- A = B->C
curry :: [\f::B] -> [ \ x -> \ y -> f (x,y) ] :: C
-- C = D->E
curry :: [\f::B] -> ( [\x::D] -> [ \ y -> f (x,y) ] :: E )
-- E = F->G

curry :: [\f::B] -> ( [\x::D] -> ( [\y::F] -> [f (x,y)] :: G )
-- B = H->G

curry :: [\f::H->G] -> ( [\x::D] ->( [\y::F] ->[f(x,y)] :: G )
-- H = D*F

curry :: [\f::D*F->G] -> ( [\x::D] ->( [\y::F] ->[f(x,y)] :: G )
-- A = B->C = B->D->E = B->D->F->G
-- = (H->G)->D->F->G = (D*F->G)->D->F->G
uncurry :: (a -> b -> c) -> (a,b)-> c
uncurry = \g -> \ (x,y) -> g x y

uncurry :: [ \g -> \ (x,y) -> g x y ] :: A
-- A = B->C
uncurry :: [\g::B] -> [ \ (x,y) -> g x y ] :: C
-- C = D*E->F
uncurry :: [\g::B] -> ( [\ (x,y)::D*E] -> [ g x y ] :: F )
-- B = D->G
uncurry :: [\g::D->G] -> ( [\(x,y)::D*E] -> [ g x y ] :: F )
-- G = E->F
uncurry :: [\g::D->E->F] -> ( [\(x,y)::D*E] -> [ g x y ] :: F
-- A = B->C = B->D*E->F = (D->G)->D*E->F = (D->E->F)->D*E->F
Parameters $A \ B \ C : \text{Set}$.

Definition curry ($f : A \times B \to C$) := fun $a \Rightarrow$ fun $b \Rightarrow f(a, b)$.
Definition uncurry ($g : A \to B \to C$) := fun $p \Rightarrow g(\text{fst} \ p) \ (\text{snd} \ p)$.

(* Prove that uncurry (curry f) is extensionally equivalent to f. *)
Theorem Left_Inverse : \forall f \ a \ b, \ \text{uncurry} \ (\text{curry} \ f) \ (a, b) = f(a, b).
Proof.
  intros.
  unfold curry, uncurry.
  simpl. (* \text{fst}(a,b)=a; \text{snd}(a,b)=b *)
  reflexivity. (* f(a,b)=f(a,b) *)
Qed.

(* Prove that curry (uncurry g) is extensionally equivalent to g. *)
Theorem Right_Inverse : \forall g \ a \ b, \ \text{curry} \ (\text{uncurry} \ g) \ a \ b = g \ a \ b.
Proof.
  intros.
  unfold curry, uncurry.
  simpl.
  reflexivity.
Qed.
**Definition** Isomorphic $(X Y : \text{Set}) :=$

exists $f : X \to Y$,

exists $g : Y \to X$,

$(\forall x, g(f(x)) = x \land (\forall y, f(g(y)) = y))$.

**Axiom** Extensionality : $\forall (X Y : \text{Set}) (f g : X \to Y)$,

$(\forall x, f x = g x) \to f = g$.

**Theorem** IsoFunSpace : Isomorphic $(A * B \to C) (A \to B \to C)$.

**Proof.**

exists curry, uncurry.

split.

apply Extensionality. intros [a b].

apply Left_Inverse. intro g.

apply Extensionality. intro a.

apply Extensionality. intro b.

apply Right_Inverse.

Qed.
Outline

1. Common Patterns
2. Function Spaces, Currying
3. Haskell Data Structures
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.
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
```
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
```

Maybe is a highly significant (and predefined) type constructor (kind $* \rightarrow *$) in Haskell. See the class `Optional<T>` introduced in Java 8.
Haskell Types

Type constructors can take types as parameters.

```haskell
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.

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]
Haskell Trees

See Hudak PPT, Ch7.

```
SimLeaf :: SimpleTree
SimBranch SimLeaf SimLeaf :: SimpleTree
SimBranch SimLeaf (SimBranch SimLeaf SimLeaf) :: SimpleTree
SimBranch (SimBranch SimLeaf SimLeaf) SimLeaf :: SimpleTree
```
Nested Types

\[
data \text{ List } a = \text{ NilL} \mid \text{ ConsL } a \ (\text{List } a)
data \text{ Nest } a = \text{ NilN} \mid \text{ ConsN } a \ (\text{Nest } \ (a,a))
data \text{ Bush } a = \text{ NilB} \mid \text{ ConsB } a \ (\text{Bush } (\text{Bush } a))
data \text{ Node } a = \text{ Node2 } a \ a \ a \mid \text{ Node3 } a \ a \ a \ a
data \text{ Tree } a = \text{ Leaf } a \mid \text{ Succ } (\text{Tree } (\text{Node } a))
\]

Hinze, Finger Trees.
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