

- *Syntax*

```
e ::= x
    | \x -> e
    | e1 e2
```

- Programs are *expressions* or λ -terms
- *Variable*: **x**, **y**, **z**
- *Abstraction*: (aka nameless function definition) $\backslash x \rightarrow e$ means “for any **x**, compute **e**”; **x** is the *formal parameter*, **e** is the *body*
- *Application*: (aka function call) **e1 e2** means “apply **e1** to **e2**”; **e1** is the *function* and **e2** is the *argument*
- *Syntactic Sugar*: convenient notation used as a shorthand for valid syntax

```
-- instead of:           we write:
\ x -> (\ y -> (\ z -> e))  \ x -> \ y -> \ z -> e
\ x -> \ y -> \ z -> e      \ x y z -> e
(((e1 e2) e3) e4)         e1 e2 e3 e4
```

- *Scope of a variable* The part of a program where a *variable is visible*
- In the expression $\backslash x \rightarrow e$
 - **x** is the newly-introduced variable
 - **e** is the *scope* of **x**
 - Any occurrence of **x** in $\backslash x \rightarrow e$ is *bound* (by the *binder* $\backslash x$)
 - An occurrence of **x** in **e** is *free* if it is **not bound** by an enclosing abstraction
- *Free Variables*: A variable **x** is *free* if there exists a free occurrence of **x** in **e** (not bound as a formal)
- *Closed Expressions*: if **e** has no free variables it is *closed*
- α -step (renaming formals): we can rename a formal parameter and replace all its occurrences in the body
- β -step (aka function call)
 - $(\backslash x \rightarrow e1) e2 \Rightarrow e1[x := e2]$
 - $e1[x := e2]$ means “**e1** with all free occurrences of **x** replaced with **e2**”
 - Computation is **search and replace**: if you see an *abstraction* applied to an argument, take the *body* of the abstraction and replace all free occurrences of the *formal* by that argument
- *Normal Forms*:
 - A *redex* is a λ -term of the form $(\backslash x \rightarrow e1) e2$
 - A λ -term is in *normal form* if it contains no redexes
- *Evaluation*:
 - A λ -term **e** evaluates to **e'** if there is a sequence of steps

```
e ==> e_1 ==> ... ==> e_N ==> e'
```

- each \Rightarrow is either $\Rightarrow a$ or $\Rightarrow b$ and $N \geq 0$
- **e'** is in normal form
- $e1 \Rightarrow^* e2$: **e1** *reduces* to **e2** in 0 or more steps
- $e1 \Rightarrow^> e2$: **e1** *evaluates* to **e2**
- Ω : $(\backslash x \rightarrow x x) (\backslash x \rightarrow x x)$
- *Recursion*: Fixpoint Combinator

```
FIX STEP
==> STEP (FIX STEP)
```

- $\text{FIX} = \backslash \text{stp} \rightarrow (\backslash x \rightarrow \text{stp } (x x)) (\backslash x \rightarrow \text{stp } (x x))$
- *Quicksort in Haskell*

```
sort :: [a] -> [a]
sort [] = []
sort (x:xs) = sort ls ++ [x] ++ sort rs
  where
    ls = [ l | l <- xs, l <= x ]
    rs = [ r | r <- xs, x < r ]
```

- *Functions in Haskell*
 - Functions are *first-class values*
 - can be *passes as arguments* to other functions
 - can be *returned as results* from other functions
 - can be *partially applied* (arguments passed *one at a time*)
- *Top-level bindings*:
 - Things can be defined globally
 - Their names are called *top-level variables*
 - Their definitions are called *top-level bindings*
- *Equations and Patterns*

```
pair x y b = if b then x else y
fst p      = p True
snd p      = p False
```

- A single function binding can have multiple equations with different *patterns* of parameters

- The first equation whose pattern matches the actual arguments is chosen
- *Referential Transparency* means that a variable can be defined *once per scope* and *no mutation is allowed*; the same function always evaluates to the same value
- *Local variables* can be defined using a **let** expression

```
sum 0 = 0
sum n = let n' = n - 1
         in n + sum n'
```

- Syntactic sugar for nested **let** expressions:

```
sum 0 = 0
sum n = let
    n'      = n - 1
    sum'    = sum n'
  in n + sum'
```

- If you need a variable whose scope is an equation, use the **where** clause instead:

```
cmpSquare x y | x > z    = "bigger :)"
               | x == z   = "same :|"
               | x < z    = "smaller :("
  where z = y * y
```

- *Types*:

- In Haskell every expression either *has a type* or is *ill-typed* and rejected at compile-time
- Types can be annotated using ::

```
haskellIsAwesome :: Bool
haskellIsAwesome = True
```

- Functions have *arrow types*
- $\backslash x \rightarrow e$ has type **A** \rightarrow **B**
- If **e** has type **B** assuming **x** has type **A**
- A *Combinator* is a function with *no free variables*
- *Lists*:
 - A list is either an *empty list*: **[]**
 - Or a *head element* attached to a *tail list*: **x:xs**

```
[]           -- A list with zero elements
1:[]         -- A list with one element
(:) 1 []     -- A list with one element
1:(2:(3:(4:[]))) -- A list with four elements
1:2:3:4:[]   -- Same thing
[1,2,3,4]    -- Syntactic sugar
```

- **[]** and **:** are called the list *constructors*
- A list has type **[A]** if each one of its elements has type **A**
- *Pairs*: the constructor is **(,)**

```
myPair :: (String, Int)
myPair = ("apple", 3)
```

- *Record Syntax*:
 - Instead of:

```
data Date = Date Int Int Int
```

- You can write:

```
data Date = Date {
    month :: Int,
    day   :: Int,
    year  :: Int
}
```

- Use the field name as a function to access part of the data:

```
deadlineDate = Date 1 10 2019
deadlineMonth = month deadlineDate
```

- Building data types:
 - *Product* types (each-of): a value of **T** contains a value of **T1** *and* a value of **T2**
 - *Sum* types (one-of): a value of **T** contains a value of **T1** *or* a value of **T2**
 - *Recursive* types: a value of **T** contains a *sub-value* of the same type **T**
- *Pattern Matching*:

```
html :: Paragraph -> String
```

```
html (Text str)      = ...
html (Heading lvl str) = ...
html (List ord items) = ...
```

- Match for arbitrary data types
- Dangers: *missing* or *overlapped* patterns
- Pattern matching expression

```
html :: Paragraph -> String
html p =
  case p of
    Text str      -> ...
    Heading lvl str -> ...
    List ord items -> ...
```

- The **case** expression has type **T** if every output expression has type **T** and the input is a valid pattern for the type; the input expression is called the *match scrutinee*
- *Tail Recursion*: The recursive call is the *top-most* sub-expression in the function body; no computations allowed on recursively-returned body; the value returned by the recursive call is the value returned by the function
- Tail-recursive factorial:

```
loop acc n
  | n <= 1 = acc
  | otherwise = loop (acc * n) (n - 1)
```

- Tail recursive calls compile to fast loops automatically
- The *Filter* pattern:

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs)
  | f x      = x : filter f xs
  | otherwise = filter f xs
```

- Higher-order function which takes function **f** and a list as arg
- For each element **x** in the list, if **f x == True** then **x** will be in the output list
- The *Map* pattern:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

- Higher order function which takes a function **f** and a list as arg
- For each element **x** in the input list, **f x** will be in the output list
- The *Fold-Right* pattern:

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

- Higher order function which recurses on the tail
- Combines result with the head in some binary operation
- **len** = `foldr (\x n -> 1 + n) 0`
- **sum** = `foldr (\x n -> x + n) 0`
- **cat** = `foldr (\x n -> x ++ n) ""`
- The *Fold-Left* pattern:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f b xs = helper b xs
  where
    helper acc [] = acc
    helper acc (x:xs) = helper (f acc x) xs
```

- Higher order function uses a helper function with an extra accumulator argument
- To compute the new accumulator, combine the urrent accumulator with the head using some binary operation
- Useful HOFs:
 - **Flip**: flips the order of the input args

```
flip :: (a -> b -> c) -> b -> a -> c
```

- **Compose**: compose functions

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- Libraries will implement **map**, **fold**, **filter**, etc on its collections
- List Comprehensions*: List comprehensions construct a new list from an old list

```
-- examples
map f xs  = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
concat xss = [x | xs <- xss, x <- xs]
```

```
-- rules
[e | True]      = [e]
[e | q]         = [e | q, True]
[e | b, Q]      = if b then [e | Q] else []
[e | p <- xs, Q] = let ok p = [e | Q]
                  ok _ = []
                  in concat (map ok xs)
```

- Functions from Homework:*

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

```
digitsOfInt :: Int -> [Int]
digitsOfInt n
  | n < 0   = []
  | n < 10  = [n]
  | otherwise = digitsOfInt (div n 10) ++ [mod n 10]
```

```
digits :: Int -> [Int]
digits n = digitsOfInt (abs n)
```

```
additivePersistence :: Int -> Int
additivePersistence n
  | n < 10   = 0
  | otherwise = 1 + additivePersistence (sumList (digitsOfInt n))
```

```
digitalRoot :: Int -> Int
digitalRoot n
  | n < 10   = n
  | otherwise = digitalRoot (sumList (digitsOfInt n))
```

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

```
let ZERO = \f x -> x
let ONE  = \f x -> f x
let TWO  = \f x -> f (f x)
let SUCC = \n f x -> f (n f x)
let EXP  = \n m -> n (MUL m) ONE

let TRUE = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2
let NOT   = \b1 -> b1 FALSE TRUE
let NOR   = \b1 b2 -> NOT (OR b1 b2)
let NAND = \b1 b2 -> NOT (AND b1 b2)
let XOR   = \b1 b2 -> AND (NAND b1 b2) (OR b1 b2)

let INCR = \n f x -> f (n f x)

let PAIR = \x y b -> b x y
let FST  = \p -> p TRUE
let SND  = \p -> p FALSE

let SKIP1 = \j k -> (\b -> b TRUE ((AND TRUE (k(TRUE))) (j(k(FALSE)))) (k(FALSE))))
let DECR  = \n -> (n (SKIP1 INCR) (PAIR FALSE ZERO)) FALSE
let SUB   = \n n -> (n DECR) m
let ISZ   = \n -> n(\a -> FALSE) TRUE
let EQL   = \n m -> AND (ISZ (SUB m n)) (ISZ (SUB n m))
let SUC   = \n f x -> f (n f x)
let ADD   = \n m -> n SUC m
let MUL   = \n m -> n (ADD m) ZERO
let REPEAT = \n m -> n (PAIR m) FALSE
let EMPTY = \p -> p (\x y z -> FALSE) TRUE
let FIX   = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
let LEN   = FIX (\rec n -> (EMPTY n) ZERO (INCR (rec (SND n))))
let STEP  = \rec n -> ITE (ISZ n) ZERO (ADD n (rec (DECR n)))
let SUM   = FIX STEP
let DIV   = FIX (\rec n m -> ITE (EQL n m) ONE (ITE (ISZ (SUB n m)) ZERO (INCR (rec (SUB n m)))))
let MOD   = FIX (\rec n m -> ITE (EQL n m) ZERO (ITE (ISZ (SUB n m)) ) n (rec (SUB n m) m)))
let INSERT = \n m -> (PAIR n m)
let APPEND = FIX (\rec n m -> ITE (EMPTY n) m (INSERT (FST n) (rec (SND n) m)))
```

Listing 1: "Lambda Calculus Functions" label