- *Syntax*

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

- Programs are *expressions* or λ-terms
- *Variable:* `x`, `y`, `z`
- *Abstraction:* (aka nameless function definition) `\x -> 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 `\x -> e`
  - `x` is the newly-introduced variable
  - `e` is the *scope* of `x`
  - Any occurrence of `x` in `\x -> e` is *bound* (by the *binder* `\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)
  - `(\x -> e1) e2 =b> 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 `(\x -> 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 `=?>` is either `=a>` or `=b>` and `N >= 0`
  - `e'` is in normal form
  - `e1 =*> e2`: `e1` *reduces* to `e2` in 0 or more steps
  - `e1 =~> e2`: `e1` *evaluates* to `e2`
- Ω: `(\x -> x x) (\x -> x x)`
- *Recursion:* Fixpoint Combinator

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

  - `FIX = \stp -> (\x -> stp (x x))(\x -> 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*
  - `\x -> e` has type `A -> 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]
```

- *Higher order functions from practice final*

```
reverse :: [a] -> [a]
reverse xs = foldl (\res x -> x : res) [] xs


absValues :: [Int] -> [Int]
absValues = map (\x -> if x < 0 then -x else x)


dedup :: [Int] -> [Int]
dedup = foldr insert []
  where
    insert x ys = x : (filter (/= x) ys)
```

- *Binary Search Trees*

```
size :: Tree->Int
size Empty = 0
size (Node_ l r) = 1 + size l + size r


insert :: Int -> Tree -> Tree
insert x Empty = Node x Empty Empty
insert x (Node y l r)
    | x == y    = Node y l r
    | x < y     = Node y (insert x l) r
    | otherwise = Node y l (insert x r)

sort :: [Int] -> [Int]
sort xs = toList (fromList xs)
  where
    fromList :: [Int] -> Tree
    fromList []     = Empty
    fromList (x:xs) = insert x (fromList xs)

toList :: Tree -> [Int]
toList Empty       = []
toList (Node x l r) = toList l ++ [x] ++ toList r


size :: Tree -> Int
size t = loop 0 [] t
  where
    loop :: Int -> [Tree] -> Tree -> Int
    loop acc []     Empty          = acc
    loop acc (t:ts) Empty          = loop acc ts t
    loop acc ts     (Node _ l r)   = loop (acc + 1) (r:ts) l
```

- *Lambda Calculus Functions*

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

TRUE   = \x y -> x
FALSE  = \x y -> y
ITE    = \b x y -> b x y
AND    = \b1 b2 -> b1 b2 FALSE
OR     = \b1 b2 -> b1 TRUE b2
NOT    = \b1 -> b1 FALSE TRUE
NOR    = \b1 b2 -> NOT (OR b1 b2)
NAND   = \b1 b2 -> NOT (AND b1 b2)
XOR    = \b1 b2 -> AND (NAND b1 b2) (OR b1 b2)
INCR   = \n f x -> f (n f x)
PAIR   = \x y b -> b x y
FST    = \p    -> p TRUE
SND    = \p    -> p FALSE
SKIP1  = \j k -> (\b ->
    b TRUE ((AND TRUE (k(TRUE))) (j(k(FALSE))) (k(FALSE))))
DECR   =  \n -> (n (SKIP1 INCR) (PAIR FALSE ZERO)) FALSE
SUB    = \m n -> (n DECR) m
ISZ    =  \n -> n(\a -> FALSE) TRUE
EQL    = \n m -> AND (ISZ (SUB m n)) (ISZ (SUB n m))
SUC  = \n f x -> f (n f x)
ADD    = \n m -> n SUC m
MUL    = \n m -> n (ADD m) ZERO
REPEAT = \n m -> n (PAIR m) FALSE
EMPTY  = \p -> p (\x y z -> FALSE) TRUE
FIX    = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
LEN    = FIX (\rec n -> (EMPTY n) ZERO (INCR (rec (SND n))))
STEP   = \rec n -> (ISZ n) ZERO (ADD n (rec (DECR n)))
SUM    = FIX STEP
DIV    = FIX (\rec n m ->
    (EQL n m) ONE ((ISZ (SUB n m) ) ZERO (INCR (rec (SUB n m) m))))
MOD    = FIX (\rec n m ->
    (EQL n m) ZERO ((ISZ (SUB n m) ) n (rec (SUB n m) m)))
INSERT = \n m -> (PAIR n m)
APPEND' = FIX (\rec n m ->
    (EMPTY n) m (INSERT (FST n) (rec (SND n) m)))

-- sets
EMPTY  = \x -> FALSE
INSERT = \n s x -> ITE (EQL n x) TRUE (s x)
HAS    = \s x -> s x
INTERSECT  = \s1 s2 x -> AND (s1 x) (s2 x)
```

- *TYPE INFERENCE*
  - Typing Rules:
    - `G |- e :: T`
    - An expression `e` has type `T` in context `G` if we can derive `G |- e :: T` using these rules
    - An expression `e` is *well-typed* in `G` if we can derive `G |- e :: T` for some type `T`
    - *ill-typed* otherwise
  - Polymorphic Types
    - `forall a . a -> a`
    - `a` is a (bound) type variable
    - also called a *type scheme*
    - We *instantiate* this scheme into different types by replacing `a` in the body with some type e.g. instantiating with `Int` gives `Int -> Int`
  - Inference with polymorphic types
    - We can derive `e :: Int -> Int` where `e` is

```
let id = \x -> x in
  let y = id 5 in
    id (\z -> z + y)
```

    - When we have to pick a type `T` for `x` we pick a **fresh type variable** `a`
    - So the type of `\x -> x` comes out as `a -> a`
    - We can **generalize** this type to `forall a . a -> a`
    - When we apply `id` the first time we *instantiate* this polymorphic type with `Int`
    - When we apply `id` the second time we *instantiate* this polymorphic type with `Int -> Int`
  - Type substitutions
    - A finite map from type variables to types: `U : TVar -> Type`
    - Example: `U1 = [a / Int, b / (c -> c)]`
    - To apply a substitution `U` to a type `T` means replace all type vars in `T` with whatever they are mapped to in `U`
    - Example: `U1 (a -> a) = Int -> Int`
  - Type inference algorithm
    - Given a context `G` and an expression `e`
    - return a type `T` such that `G |- e :: T`
    - or report a type error if `e` is ill-typed in `G`
    - Depending on what kind of expression `e` is, find a typing rule that applies to
    - If the rule has premises, recursively call `infer` to obtain the types of sub-expressions
    - Combine the types of sub-expressions according to the conclusion of the rule
    - If no rule applies, report a type error
  - Constraint-based type inference
    - Whenever you need to 'guess' a type, don't. Just use a *fresh* type variable
    - Whenever a rule imposes a constraint on a type, try to find the right substitution for the free type variables to satisfy the constraint
  - Unification
    - The *unification* problem: given two types `T1` and `T2`, find a type substitution `U` such that `U T1 = U T2`
    - Such a substitution is called a *unifier* of `T1` and `T2`
  - Generalization and INstantiation:
    - Whenever we infer a type for a let-defined variable, generalize
    - Whenever we see a variable with a polymorphic type, instantiate it with a fresh type variable
- *Formalizing Nano*
  - Want to guarantee properties about programs such as: all programs terminate, evaluations is deterministic, certain programs never fail at run time, etc
  - Operational Semantics
    - Defines how to execute a program step by step
    - A *step relation* or *reduction relation* `e => e'` as: "expression `e` makes a step to an expression `e'`"
    - Defined inductively through a set of rules

```
           e1 => e1'       -- premise
[Add-L] -------------
       e1 + e2 => e1' + e2 -- conclusion
```

    - A reduction is *valid* if we can build its *derivation* by "stacking" the rules
  - Normal forms:
    - There are no reduction rules for `n` or `x`
    - Both of these expresssions are normal forms already; however, `n` is a *value* and `x` is *not a value*
    - Thus is we reduce to just `x` the program is *stuck*
  - Evaluation relation:
    - Like in λ-calculus we define the *multi-step reduction* relation `e =*> e'` iff there exists a sequence of expressions `e1, ... en` such that `e = e1`, `en = e'`, and `ei => e(i+1)` for each `i in 0..n)`
    - The evaluation relation `e =~> e'` iff `e =*> e'` and `e'` is in normal form