- *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*
- *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'
```

-

Var	Desc
B	the number of data pages
R	number of records per page
D	average time to read or write a disk page
F	average fanout for a non-leaf page

	()	Scan	Equality	Range	Insert	Delete
Heap		BD	$0.5BD$	BD	$2D$	Search + D
Sorted		BD	$D\log_2 B$	$D(\log_2 B + \#$ matching pages)	Search + BD	Search + BD
Clustered		$1.5BD$	$D\log_F 1.5B$	$D(\log_F 1.5B + \#$ matching pages)	Search + D	Search + D
Unclust. Tree		$BD(R + 0.15)$	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B + \#$ matching pages)	Search + $2D$	Search + $2D$
Unclust. Hash		$BD(R + 0.125)$	$2D$	BD	Search + $2D$	Search + $2D$