Haskell is very similar to ML, but with one important difference. ML (like Scheme) uses pass-by-value parameters, but Haskell uses pass-by-name or pass-by-text.

Instead of evaluating an argument before it is passed, we just pass the argument unevaluated.

e.g. in Lambda calculus

$$(\lambda x.\lambda y.\ x\ y)\ (\lambda z.z)\ ((\lambda w.w)\ a)$$

Before we apply the first function, let's evaluate the arguments. The 2nd. argument reduces to a. This is bound to y; x is bound to $(\lambda z.z)$. So the application reduces to

$$(\lambda z.z)\ a$$
$$= a$$

But there is another to do the main application.
Instead we pass the arguments unevaluated
x will be bound to $(\lambda z.z)$; y will be bound
to $((\lambda w.w)\ a)$

So the application reduces to

$$(\lambda z.z)\ ((\lambda w.w)\ a)$$

We can do this again — z is bound to $((\lambda w.w)\ a)$
So this reduces to $((\lambda w.w)\ a) = a$

We got the same answer doing the reduction 2 different ways. There is a theorem that says if a reduction is possible then the 2 ways produce identical results.

The first way is called "applicative order reduction".
The second is called "normal order reduction".

In PL, the 1st is pass-by-value;
the 2nd is pass-by-name.

In functional languages, the 1st is called "eager" evaluation; the 2nd is called "Lazy" evaluation.

[There are expressions which do not reduce:

$$(\lambda y.a)((\lambda x.x\ x)(\lambda x.x\ x))$$

If we try to evaluate the argument we will never finish. However, with lazy evaluation, we just get a.]

So Haskell uses lazy evaluation of parameters.

This gives facilities that ML can never have.

We can express infinite computations:

$$\text{numsFrom } n = n \underset{\text{cons}}{:} \text{numsFrom } (n+1)$$

e.g. numsFrom 0 => [0, 1, 2 .....]

We can actually use this in:

squares = map (^2) (numsFrom 0)

squares => [0, 1, 4, 9, ...]

Now define:

take 0 _ = []

take n (x:xs) = x : take (n-1) xs

Now we can do:

e.g. take 2 [1,2,3,4] => [1,2]

Now:

take 4 squares => [0,1,4,9]

We don't evaluate squares, but pass it unchanged to take.

take 4 squares

$= 0 : (\text{take } 3\ [1,4,9 \ldots])$

$= 0 : 1 : (\text{take } 2\ [4,9,\ldots])$

$= 0 : 1 : 4 : (\text{take } 1\ [9,\ldots])$

$= 0 : 1 : 4 : 9 : (\text{take } 0\ [16,\ldots])$

$= 0 : 1 : 4 : 9 : [\,]$

$= [0,1,4,9]$

We have use the infinite computation to yield one number at a time <u>when it's needed</u>.

We can have infinite data structures — list comprehensions

$$\text{squares} = [n * n \mid n \leftarrow \underbrace{[0\,..]}_{\text{infinite data structures}}]$$

again take 4 squares $\Rightarrow [0,1,4,9]$

also member 16 squares => true

however member 15 squares never stops because member searches endlessly in the list of squares.