The whole of Haskell can be translated into into a simple language called core Haskell which is a slightly enriched form of lambda calculus.

This form of lambda calculus is then implemented.

Example:

```
  length [] = 0
  length (x:xs) = 1 + length xs

  {push patterns to the right}

  length l = case l of
                [] -> 0
                (x:xs) -> 1 + length xs

  {replace multiple alts by (pattern, _) pairs

           = case l of
                [] -> 0
                _  -> case l of
                           (x:xs) - 1 + length xs
                            _ -> error ``insufficient patterns''

  {0-ary constructor rule}

           = if (l == []) then 0
             else case l of
                       (x:xs) - 1 + length xs
                       _ -> error ``insufficient patterns''

  {refutable pattern matching}

= if (l == []) then 0
             else unpack_s_: (\x\xs -> -> 1 + length xs) l)
               [] error

      where the senantics of unpack_s_: is

             unpack_s_: f (x : xs) = f x xs
             unpack_s_: f (k x xs) = FAIL, if k /= :
```

```
            unpack_s_: _|_ = _|_

    and the semantics of [] is

            v1 [] v2 = if v1 == fail then v2 else v1

{remove recursion}

 length  = Y  (\len \l -> if (l == []) then 0
                 else unpack_s_: (\x\xs -> -> 1 + len xs) l)
               [] error)
```

```
data Bintree a = Node (Bintree a) (Bintree a) | Leaf a

reflect (Leaf n) = Leaf n
reflect (Branch t1 t2) = Branch (reflect t1) (reflect t2)

{push patterns to the right}

reflect t = case t of
                 (Leaf n) -> Leaf n
                 (Branch t1 t2) -> Branch (reflect t1) (reflect t2)

{replace multiple alts by (pattern, _) pairs

          = case t of
                 (Leaf n) -> Leaf n
                 _        -> case t of
                                 (Branch t1 t2) -> Branch (reflect t1) (reflect t2)
                                 _ -> error

{refutable pattern matching}

          = unpack_s_Leaf (\n -> Leaf n) t []
            case t of
                 (Branch t1 t2) -> Branch (reflect t1) (reflect t2)
                 _ -> error

{refutable pattern matching}
```

```
          = unpack_s_Leaf (\n -> Leaf n) t []
            unpack_s_Branch (\t1\t2 -> Branch (reflect t1) (reflect t2)) t []
            error

{remove recursion}

reflect = Y (\r\t -> unpack_s_Leaf (\n -> Leaf n) t []
                     unpack_s_Branch (\t1\t2 -> Branch (r t1) (r t2)) t []
                     error)
```

```
f ~(x,y) = x

{push patterns to the right}

f z = case z of
           ~(x,y) -> x
           _ -> error

{~ pattern matching}

    = (\x'\y' -> x') (case z of (x,y) -> x) (case z of (x,y) -> y)

{error introduction in case}

    = (\x'\y' -> x') (case z of
                            (x,y) -> x
                            _ -> error)
                     (case z of
                            (x,y) -> y
                            _ -> error)

{refutable pattern matching involving product constructors}

    = (\x'\y' -> x') (unpack_p_pair (\x\y -> x) z) (unpack_p_pair (\x\y -> y) z)

unpack_p_pair f (Pair a b) = f a b
unpack_p_pair f _|_ = _|_
```

```
f ~[x,~(a,b)] -> x
```

```
{push patterns to the right}

    f l = case l of
                ~[x,~(a,b)] -> x
                _ -> error

{~ pattern matching}

        = (\x'\a'\b' -> x') (case l of [x,~(a,b)] -> x)
                            (case l of [x,~(a,b)] -> a)
                            (case l of [x,~(a,b)] -> b)

{error introduction in case}

        = (\x'\a'\b' -> x') (case l of [x,~(a,b)] -> x; _ -> error)
                            (case l of [x,~(a,b)] -> a; _ -> error)
                            (case l of [x,~(a,b)] -> b; _ -> error)

{translating case l of [x,~(a,b)] -> x; _ -> error}

       case l of
           [x,~(a,b)] -> x
           _ -> error

{reduce the pattern to one involving variables only}

    =  case l of
          (y:ys) -> case y of
                       x -> case ys of
                                ~(a,b) -> x
                                _ -> error
                       _ -> error
          _ -> error

{remove error clause while matching with variables}

    =  case l of
          (y:ys) -> case y of
                       x -> case ys of
                                [~(a,b)] -> x
                                _ -> error
          _ -> error
```

```
{pattern matching with variables}

     =  case l of
            (y:ys) -> (\x - > case ys of
                                   [~(a,b)] -> x
                                   _ -> error) y
            -> error

 {refutable pattern matching}


     =  unpack_s_: (\y\ys -> (\x - > case ys of
                                   [~(a,b)] -> x
                                   _ -> error) y) l
        [] error

 {case ... error is translated as}

     = case ys of
          (z:zs) -> case z of
                      ~(a,b)-> case zs of
                                  [] -> x
                                  _ -> error
           _ -> error

 {refutable pattern matching}
     =  unpack_s_: (\z\zs -> case z of
                             ~(a,b)-> case zs of
                                         [] -> x
                                          _ -> error) ys
        [] error

 {~pattern matching}
     =  unpack_s_: (\z\zs -> (\a'\b' -> case zs of
                                         [] -> x
                                          _ -> error)
                            (case z of (a,b) -> a)
                            (case z of (a,b) -> b)) ys
        [] error

 { refutable pattern matching, 0-ary constructors}

 {~pattern matching}
```

```
    =  unpack_s_: (\z\zs -> (\a'\b' ->
                              if zs == [] then x else error)
                            (case z of (a,b) -> a)
                            (case z of (a,b) -> b)) ys
        [] error



{~pattern matching}
    =  unpack_s_: (\z\zs -> (\a'\b' ->
                              if zs == [] then x else error)
                            (unpack_p_pair (\a'\b' -> a') ys [] error)
                            (unpack_p_pair (\a'\b' -> a') ys [] error)) ys
        [] error

{ unpack_p_pair

{~pattern matching}
    =  unpack_s_: (\z\zs -> (\a'\b' ->
                              if zs == [] then x else error)
                            (unpack_p_pair (\a'\b' -> a') ys [] error)
                            (unpack_p_pair (\a'\b' -> a') ys [] error)) ys
        [] error


{unpack_p_x _ _ [] e = unpack_p_x _ _}

    =  unpack_s_: (\z\zs -> (\a'\b' ->
                              if zs == [] then x else error)
                            (unpack_p_pair (\a'\b' -> a') ys)
                            (unpack_p_pair (\a'\b' -> a') ys)) ys
        [] error

-----------------------------------------------------------------------

let
   length [] = 0
   length (x:xs) = 1 + length xs
in length [1,2,3]


{remove patterns and recursion}
```

```
let
   length  = Y  (\len \l -> if (l == []) then 0
                  else unpack_s_: (\x\xs -> -> 1 + len xs) l)
                [] error
in length [1,2,3]

{translate to a lambda}

(\length -> length [1,2,3])
    (Y  (\len \l -> (if (l == []) then 0
                    else unpack_s_: (\x\xs -> -> 1 + len xs) l)
                    [] error))
```

```
let
  x = 1 : y
  y = 2 : x
in x

{convert mutual recursion to self-recursion}

let
  (x,y) = (1 : y, 2 : x)
in x

{remove recursion}

let
  (x,y) = (\~(x,y) -> (1 : y, 2 : x)) (x, y)
in x

let
  (x,y) = Y (\~(x,y) -> (1 : y, 2 : x))
in x

{convert let into case}

case Y (\~(x,y) -> (1 : y, 2 : x)) of
  (x,y) -> x

{refutable pattern matching with product constructors}
```

```
unpack_p_Pair (\x'\y' -> x) (Y (\~(x,y) -> (1 : y, 2 : x)))
```

This is the translation. To see that it works,

denote `\~(x,y) -> (1 : y, 2 : x)` as F. Then notice that

```
   Y F
= F (Y F)
= \~(x,y) -> (1 : y, 2 : x) (Y F)
= (\x'\y' -> (1 : y', 2 : x')) (case Y F of (x,y) -> x) (case Y F of
(x,y) -> y)
 = (1 : (case Y F of (x,y) -> y), 2 : (case Y F of (x,y) -> x))
```

Therefore

```
  unpack_p_Pair (\x'\y' -> x) (Y (\~(x,y) -> (1 : y, 2 : x)))
= (1 : (case Y F of (x,y) -> y)
= (1: 2: (case Y F of (x,y) -> x)))
= (1: 2: 1: (case Y F of (x,y) -> y))
  ...
```

Handling list comprehensions

```
[x + y | x <- [1,2,3], y <- [4,5]]
```

For every value in [1,2,3] compute x + y for all values of y.

```
[[x + y | y <- [4,5]] | x <- [1,2,3]]

= [[5,6],[6,7],[7,8]]
```

But this is not what we wanted. This can be fixed with:

```
concat [[x + y | y <- [4,5]] | x <- [1,2,3]]
```

So, tentatively, the rule is

```
[e | qual1, qual2] = concat [[e | qual2] | qual1]
```

But does the rule work if one of the qualifiers is a guard instead of a generator.

```
[sq x | x <- [1,2,3], odd x]
```

```
concat [[sq x | odd x] | x <- [1,2,3]]
```

Now what meaning should we ascribe to [sq 1 | odd 1]? Clearly, it should be a list, in fact the list [1].

So we have [[1], [sq 2| odd 2] [9]]. This indicates that [sq 2| odd 2] should be the list [].

Therefore we have

```
[e | gd] = if gd then [e] else []
```

and the earlier rule involving two qualifiers is confirmed.

Now consider

```
[sqr x | x <- [1, 2, 3]]
```

Clearly, this is the same as

```
map (\x -> sqr x) [1, 2, 3]
```

So, in general we expect to have the rule:

```
[e | p <- l] = map f l where f p = e
```

However for the example

```
[x | (3,x) <- [(1,4), (3,5), (3,9)]]
```

we get _|_ since f (1,4) in f (3,x) = x is _|_

There for we must modify the rule to

```
[e | p <- l] = map f l where f p = e
                             f _ = ?
```

We can get around the problem by having

```
[e | p <- l] = concat (map f l) where f p = [e]
```

```
                                        f _ = []
```

Therefore the final rules are:

```
[e | gd] = if gd then [e] else []
[e | p <- l] = let ok p = [e]
                   ok _ = []
                in concat (map ok l)
[e | qual1, qual2] = concat [[e | qual2] | qual1]
```