- How to convert a lambda expression into SCs

This process is called lambda lifting.

Example 1:   (\x -> x + \y -> x + y) 3

1. Take out the free variable of the inner lambda expression as a parameter

```
\x\y -> x + y
```

2. Give it a name:

```
$XY  x y = x + y
```

3. Write the original lambda term in terms of $XY

```
\x -> x + $XY x
```

4. This is already in SC form. Give this a name

```
$X x = x + $XY x
```

Summing up, we have

```
$XY x y = x + y
$X x = x + $XY x
$prog = $X 3
```

graph and instantiating code for $X are:

```
     \x           1. Create a new application node
      |           2. Create a new application node
      @           3. Make the left child of @-2 point to 1
     / \          4. Make a new application node.
    @   x         5. Make the left child of @-4 point to $XY
   / \            6. Make the right child of @-4 point to the
1 +   @              argument
     / \          7. Make the right child of @-2 point to
2  $XY  x            @-4
```

```
                    8. Make the left child of @-1 point to @-2
                    9. Make the left child of @-1 point to arg

  graph and instantiating code for $X are:

        \x            1. Create a new application node.
         |            2. Create a new application node.
        \y            3. Make the left child of @-2 point to 1
         |            4. Make the right child of @-2 point to arg1
         @            5. Make the left child of @-1 point to @-2
        / \           6. Make the right child of @-1 point to arg2
       @   y
      / \
  1  +   x

- Example 2:   (\x -> x + \y -> f  (\z -> x + y + z)) 6

  1. Take the innermost lambda abstraction and bring out free
     variables as parameters
        \x\y\z -> x + y + z
  2. Name the SC
        $XYZ x y z = x + y + z
  3. Substitute back
        (\x -> x + \y -> f ($XYZ x y)) 6
  4. Do this for the next abstraction
        $Y x y = f ($XYZ x y)
  5. Substitute back
        (\x -> x + $Y x) 6
  6. $X x = x + $Y x

  7. $Prog = $X 6

     $XYZ x y z = x + y + z
     $Y x y = f ($XYZ x y)
     $X x = x + $Y x
     $Prog = $X 6

- Example 3:   (\x \y -> f  \z -> x + y + z) 6

   - If we take out the free variables of \z -> x + y + z as
     \x\y\z -> x + y + z, we get

     $XYZ x y z = x + y + z
```

```
    $XY x y = f ($XYZ x y)
    $Prog = $XY 6
```

Recursive Supercombinators

- If we permit cycles in the graph, we can retain let(rec)s. There is a direct correspondence between letrecs and cyclic graphs.

  A let corresponds to a possibly cyclic graph

```
  let
     fact = \n -> if n == 0 then 1 else n * fact(n-1)
  in fact 6
```

 is converted to a graph

```
                  @
                 / \
              # \n
                |
                @
              ....
                      @
                     / \
                    #  n-1
```

 This is more efficient than using Y explicitly. If we use Y, then after two reductions we get this graph

 A graph can always be converted to a letrec

```
                @
               / \
              /   @---|
             @___/ \  |
            / \     3 |
           f   --------
```

 can be written as

```
let
   a = b c
   b = f c
   c = b 3
in a
```

- However, pattern matching, case etc must still be transformed.

- A letrec at the outermost level will be converted into recursive supercombinators. Inner letrecs will be left as such.

- After lambda lifting, the general form of the program is

```
$X1 a b c = e1
$X2 x y z = e2

$Prog = e
```

Here e1 e2 e etc may contain letrecs

Example:

```
let
  sumInts m = sum (count 1)
              where
                 count n | n>m=[]
                         | otherwise=n:count(n+1)
  sum [] = 0
  sum (x:xs) = n+sum ns
in  sumInts 100
```

- After conversion to the form we want:

```
letrec
   sumInts = \m. letrec
                    count = \n->if (n>m) then [] else n:count(n+1)
                 in sum (count 1)
   sum = \l-> if l==[] then 0 else (head l)+sum(tail l)
in
   sumInts 100
```

- Lambda lift the rhs of count

```
$N count m n  = if (n>m) then [] else n:count(n+1)
```

- rewrite

```
letrec
   sumInts = \m. letrec
                    count = $N count m
                 in sum (count 1)
   sum = \l-> if l==[] then 0 else (head l)+sum(tail l)
in
   sumInts 100
```

- Now there are no more inner lambdas. Therefore we can write sumInts and sum as recursive supercombinators.

```
$sumInts m = letrec
               count = $N count m
             in sum (count 1)
$sum l == if l==[] then 0 else (head l)+sum(tail l)
$Prog = sumInts 100
```

Points about lambda lifting
----

1. All abstractions are to be lifted to the top level as supercombinators.

2. The final program, after lambda lifting, should be of the form

```
$X arg1 arg2 ... argn = e1
$Y arg1 arg2 ... argm = e2
$Prog
```

3. The ei's should not involve any lambda abstractions. Which means, they are formed of arguments, supercombinators and letrec defined variables combined by application.

Example

```
     Method 1
     --------
     sumInts = \m. letrec
                      count = \n->if (n>m) then [] else n:count(n+1)
                   in sum (count 1)
     sum = \l-> if l==[] then 0 else (head l)+sum(tail l)
  in
     sumInts 100
```

The main problem is the lambda abstraction

```
\n->if (n>m) then [] else n:count(n+1)
```

There are two free variables in it, count and m. So bring them out as parameters:

```
$countmn count m n = if (n > m) then [] else n:count(n+1)
```

Substituting this back we get:

```
$countmn count m n = if (n > m) then [] else n:count(n+1)
sumInts = \m. letrec
                 count = $countmn count m
```

```
                 in sum (count 1)
   sum = \l-> if l==[] then 0 else (head l)+sum(tail l)
in
   sumInts 100
   ---------------------------------------------------------
```

The other functions are already in supercombinators form. So we get:

```
   $countmn count m n = if (n > m) then [] else n:count(n+1)
   $sumInts m =  letrec
                     count = $countmn count m
                 in $sum (count 1)
   $sum l = if l==[] then 0 else (head l)+$sum(tail l)

   $Prog = $sumInts 100
```

Method 2:

In this method we do not take out the letrec bound variable

```
   count' = \m\n -> if (n > m) then [] else n:count(n+1)
```

But since the body of count' contains the free variable count, we cannot lift it out of the inner letrec.

```
   sumInts = \m. letrec
                     count' = \m\n -> if (n > m) then [] else n:count(n+1)
                     count = count' m
                 in sum (count 1)
   sum = \l-> if l==[] then 0 else (head l)+sum(tail l)
in
   sumInts 100
   -----------------------------------------------------------
```

Now replece count (n+1) by count' m (n+1) in the body of count'. Now count' does not have any free variable and therefore can be lifted to the top as a supercombinator:

```
   $count' m n = if (n > m) then [] else n:count' m (n+1)
   sumInts = \m. letrec
                     count = count' m
```

```
                 in sum (count 1)
   sum = \l-> if l==[] then 0 else (head l)+sum(tail l)
in
   $sumInts 100
```

Now we can replace the inner letrec by sum ($count' m 1). With some more cleanup, we get

```
$count' m n = if (n > m) then [] else n:count' m (n+1)
$sumInts m = $sum ($count' m 1)
$sum l = if l==[] then 0 else (head l) + $sum(tail l)
$Prog = $sumInts 100
```

FULLY LAZY LAMBDA LIFTING

Consider the function

```
 f = \y -> + y (sqrt 4)
```

f is a supercombinator. However, if this lambda abstraction is shared, the graph corresponding to (sqrt 4) will be created twice.

```
              @  <- as soon as we come to this node, we
             / \    create a duplicate
          sqrt  4
```

Now this is a 'constant expression' which can be detected statically. In the context of interpretation, we could not prevent the creation of such duplicates, because we did not want to look too far below down the graph. One might complain that while compiling, we might be willing to do so and avoid duplication.

The point is that such constant expressions might get created on the fly during execution.

```
let
  f = g 4
  g x y = y + sqrt x
in
  f 1 + f 2
```

Let us trace through the execution of this example:

```
           @
         /  \
        @    @
       / \  / \
      +   @#   2
         / \
       #f   1
```

```
    @
   /  \
  @    @
 / \  / \
+   @#   2
   / \
 #@   1
 / \
g   4
```

```
    @
   /  \
  @    @
 / \  / \     #@
+   @#   2    / \
   /  \      g   4
  @    @
 / \  / \
+   1sqrt4
```

```
    @
   /  \
  @    @
 / \  / \     #@
+   @#   2    / \
   / \       g   4
  @   2
 / \
+   1
```

sqrt 4 has been evaluated once. It will be evaluated once again when g 4 2 is evaluated the second time.

A reduction is fully lazy if every expression is evalated at most once after all its free variables have been bound.

Consider the same example once again:

```
 let
   f = g 4
```

```
   g x y = y + sqrt x
 in
   f 1 + f 2
```

While lambda lifting we shall consider one lambda abstraction at a time.

```
   \y-> y + sqrt x  <- maximal free expression
            ------
```

Lambda lift a maximal free expression

```
   $sqrtxy sqrtx y = y + sqrtx
```

Substitute back:

```
 $sqrtxy sqrtx y = y + sqrtx
 let
   f = g 4
   g x = $sqrtxy (sqrt x)
 in
   f 1 + f 2
```

Now lift the remaining functions, which are supercombinators, to the top level.

```
 $sqrtxy sqrtx y = y + sqrtx
 $f = $g 4
 $g x = $sqrtxy (sqrt x)
 $Prog = $f 1 + $f 2
```

Now the reduction sequence will be:

```
            @
          /  \
         @    @
        / \  / \
       +   @#   2
          / \
        #@   1
        / \
       g   4
```

```
            @
          /  \
         @    @
        / \  / \
       +   @#   2
          / \
        #@   1
        / \
$sqrtxy   @
          / \
       sqrt  4
```

```
            @
          /  \
         @    @
        / \  / \
       +   @#   2      @#
          / \         / \
        @    @*  $sqrtxy *
       / \  / \
      +   1sqrt4
```

As can be clearly seen, the same copy of (sqrt 4) will be shared by both reductions.

Main point: Abstract out a 'maximal free expression' instead of a variable.

Example:

```
let
   f = \x -> letrec
               fac = \n -> if n==0 then 1 else n*fac(n-1)
             in + x (fac 1000)
in + (f 3) (f 4)
```

Take the \n -> abstraction and take out the free variable fac from it.

Lift it to the top>

```
$facn fac n = if n==0 then 1 else n*fac(n-1)
let
   f = \x -> letrec
               fac = $facn fac
             in + x (fac 1000)
in + (f 3) (f 4)
```

But fac 1000 will be computed twice. So take it out to the top level. This cannot be done unless one takes out fac also to the top level.

```
$facn fac n = if n==0 then 1 else n*fac(n-1)
$fac = $facn $fac
$fac1000 = $fac 1000
$f x = + x $fac1000
$Prog = + ($f 3)  ($f 4)
```