GRAPH REDUCTION
----- ---------

Summary:

- Any functional program can be represented in terms of an enriched
  lambda term

- This lambda term can be represented in the form of graph.

  Examples of

  - f (1+3)
  - \x\y\z-> x z (y z)
  - (x : xs)

- The lambda term represented as a graph is reduced to WHNF by repeatedly
  - finding the next redex
  - reducing it

- How is the next redex found?

  - Some lambda terms and their graph representations:

  1. (\x->e1)((\y->e2) e3)

```
      @
     / \
   \x    @
    |   / \
   e1 \y e3
       |
       e2
```

  2. ((\x->e1)((\y->e2) e3)) ((\x->e1)((\y->e2) e3))

```
             @
          /     \
        /         \
      @             @
     / \           / \
```

```
    \x    @      \x    @
     |   / \      |   / \
     e1 \y e3     e1 \y e3
         |            |
         e2           e2
```

```
3. ((5+3) + 2)

             @
            / \
           @   2
          / \
         +   @
            / \
           @   3
          / \
         +   5
```

```
- In general the expression will have a form

  e e1 e2 e3 e4   en

  represented as

             @
            . \                     Spine
           .   en                   tip of the spine
          .                         unwinding the spine
         @                          vertebrae
        / \                         ribs
       @   e2
      / \
     e   e1

    e may be a number or : - already in WHNF
    e may be a built in taking k arguments
               - k > n - already in WHNF
               - k <= n - (e e1 e2 ... ek) is the redex
```

After having identified the redex, we must do the reduction

```
            @     =>   @
           / \        / \
         \x True    not True
          |
          @
         / \
       not  x
```

There are three issues here:

```
(i)         @
           / \
         \x  /\$
          |  --
          @
         / \
       not  x
```

```
   while doing the substitution, two possibilities
      (i)  copy the argument tree at the position of x
      (ii) make the pointer to x now point to the tree
   we do (ii) to get
```

```
            #@
           / _ \
         \x /  /\$
          |/   --
          @
         /
       not
```

(ii) Now the result is written to the root of the redex so that
     sharing is enabled

```
            #@
           /  \
         not  /\$
               --
```

(iii) The above picture may be incorrect. Since the lambda abstraction may be shared, it must not be destroyed during instantiation.

```
           #@        =>        #@        \x
```

```
      / \               /  \       |
    \x  /\$          not  /\$      @
     |  --                --      / \
     @                         not   x
    / \
  not  x
```

This can be implemented by the following simple algorithm

```
 instantiate(body, var, value)

 - if body is a variable x and var = x then return value
 - if body is a variable x and var <> x then return body
 - if body is a constant or built in function, return body
 - if body is (e1 e2), then return the application
      instantiate(e1, var, value)  instantiate(e2, var, value)
 _ if body is \x.e and x = var then return body
 _ if body is \x.e and x <> var then return \x.instantiate(e, var, value)
```

The root of the redex is updated with whatever is returned by
instantiate(body, var, value)

```
Consider the following examples:

 - Example 1: (\x -> not x) t

      1@      instantiate(4, x, 3)
      / \
  2 \x   /\3
     |   --
   4 @
    / \
5 not  x 6


instantiate(4, x, 3)

=                      @7
                     /   \
                    /     \
      instantiate(5,x,3) instantiate(6,x,3)

=                      @7
                     /   \
                    /     \
                 5 not     3


 - Example 2: (\x\x -> 2 + x) t

          1@      instantiate(*, x, 4) = 4 \x
          / \                              |
      2 \x   /\3                       5   @
          |   --                          / \
        4\x                          6  @   2 7
          |                            / \
        5 @                        8  +   x 9
         / \
      6 @   2 7
       / \
    8 +   x 9


 - Example 3: (\x\y -> 2 + x) t
```

```
         1@       instantiate(*, x, 4) = 10 \y
         / \                               |
      2 \x   /\3                        instantiate(5,x,3)
         |   --
       4\y
         |                     =  10 \y
      5  @                            |
        / \                       11  @
    6  @   2 7                      /   \
      / \          instantiate(6,x,3)    instantiate(7,x,3)
  8  +   x  9

                               =   10 \y
                                       |
                                   11  @
                                     /   \
                                 12 @    2
                                   / \
                                8 +   3
```

```
- Example 4: (\x -> x) t

          1@          instantiate(4, x, 3) =  t 3
          / \
       2 \x  t 3
          |
       4  x
```

```
- Example 5:

         1  @
           / \        =>          10 @
      2  \x   t 3                   / \
          |                     11 @   t 3
      4   @                       / \
         / \                  7  +  (3+5) 12
     5  @   x 6
       / \
    7 +  (3+5) 8
```

So if the lambda abstraction is shared by two applications, the expression (3+5) is computed twice. So this method is not fully lazy.

Implementing Y

Use one of the following rewrite rules

```
 1.         1  @     =>     1 @
             / \             / \
            Y   f           f   @
                               / \
                              Y   f


                               --
                              |  |
                              v  |
 2.         1  @     =>     1 @--
             / \             /
            Y   f           f
```

Exercise Show the steps in the graph reduction of

```
       (Y \f\n-> if n == 0 then 1 else n * f (n - 1)) 3
```

TOWARDS COMPILATION

The instantiation process is an inefficient one-

```
             @
            / \
          \x   t
           |
         5 @
          / \
       6  @  2 9
        / \
     7 +   x 8
```

```
 While instanting the detailed steps are:

 1. Check the type of node 5
     - @ node
 2. Make a new @ node

 3. Check the type of node 6
     - @ node
 4. Make a new @ node
 5. Check the type of node 7
     - constant +
 6. Make the left child of @-4 point to
    node 7
 7. Check the type of node 8.
     - var
 8. Is this also the lambda variable?
     - yes
 9. Make the right child of @-4 point to
    the location of the argument t
11. Check the type of node 9
    constant
10. Make the left-child of @-3 point to
    the node created in 4
12  Make the right-child of @-3 point to
    the node

Every time (\x.x+2) is instantiated, this process is repeated

We can, instead separate the checking parts with the imperative
parts. The idea is that the checking part is done once and code
is generated to do the imperative part. This code can be executed
as many times as required.

 2. Make a new @ node
 4. Make a new @ node
 6. Make the left child of @-4 point to
    node 7
 9. Make the right child of @-4 point to
    the location of the argument t
10. Make the left-child of @-3 point to
    the node created in 4
12  Make the right-child of @-3 point to
```

```
   the node



 Instantiating the lambda body requires us to know its shape at
compile time. This does not happen if the lambda variable has
free variables as in

      \z -> z + (\y -> y + z) t

code cannot be generated at compile time to instantiate \y -> y +
z because the instantiation will depend on the value of z.
       - different situation than making a pointer point to an
         argument.

 Situation can be handled if there is a continuous run of lambdas
      \z\y -> y + z
 Here we do simultaneous instantiation of z and y
       - we can change the definition of WHNF to say that
         \z\y -> y + z applied to less than two arguments
         is in WHNF


 So what sort of lambda terms can be compiled?  Answer -
 supercombinators.

 A supercombinator, $S, of arity n is a lambda expression of the
 form \x1 \x2 ...\xn -> E such that
 (i) E is not a lambda expression
(ii) $S has no free variables
(iii) any lambda abstraction in E is a supercombinator
(iv) n >= 0, i.e. there may not be lambdas at all

 Examples of SCs - 3, (+ 2 5), +2, \x.x, \x.x+1
                  \x\y. x - y, \f.f (\x\y. x - y)

 Examples of non-SCs - \x.y, \y.x-y, \f.f(\x f x 3)

 A SC of arity n is not necessarily a function of n arguments.
 They will be involved in instantiation involving n arguments.
```