```
<html class="gr__web_engr_oregonstate_edu"><head><style type="text/css"></style></head><body
data-gr-c-s-loaded="true"><pre style="word-wrap: break-word; white-space: pre-wrap;">Note: This
file can be directly loaded into ghci.


EXERCISE 1 [20+15 = 35 Points]

The abstract syntax of the the stack language was given in
the exercise.

&gt; type Prog = [Cmd]
&gt;
&gt; data Cmd = LD Int | ADD | MULT | DUP | INC | SWAP | POP Int
&gt;            deriving Show

The rank of a stack is given by an integer, and the rank of an
operation is given by a pair of integers.

&gt; type Rank    = Int
&gt; type CmdRank = (Int,Int)

(a)
The type checker for a stack program starts with an initial rank of 0
and transforms this rank using the rank information for each
operation.

&gt; rankC :: Cmd -&gt; CmdRank
&gt; rankC (LD _)  = (0,1)
&gt; rankC ADD     = (2,1)
&gt; rankC MULT    = (2,1)
&gt; rankC DUP     = (1,2)
&gt; rankC INC     = (1,1)
&gt; rankC SWAP    = (2,2)
&gt; rankC (POP k) = (k,0)

The rank of a program can be computed as follows.

&gt; rankP :: Prog -&gt; Maybe Rank
&gt; rankP cs = rank cs 0

&gt; rank :: Prog -&gt; Rank -&gt; Maybe Rank
&gt; rank []     s = Just s
&gt; rank (c:cs) s | s&lt;n       = Nothing
&gt;               | otherwise = rank cs (s-n+m)
&gt;                 where (n,m) = rankC c


(b)
The semantic function can be simplified by removing the Maybe data
type. Static type checking prevents any possible stack errors.

semStatTC :: Prog -&gt; Maybe Stack
semStatTC p = case rankP p of
                Nothing -&gt; Nothing
                Just r  -&gt; Just (sem [] p)  -- sem cannot fail

The function sem can be implemented with the following type.

sem :: Prog -&gt; Stack -&gt; Stack
sem ... = ...



EXERCISE 2 [15+20 = 35 Points]


(a)
Types as given in the exercise.

&gt; data Shape = X
&gt;            | TD Shape Shape
```

```
&gt;            | LR Shape Shape
&gt;            deriving Show
&gt;
&gt; type BBox = (Int,Int)
```

Bounding boxes can be computed as follows.

```
&gt; bbox :: Shape -&gt; BBox
&gt; bbox X           = (1,1)
&gt; bbox (TD s1 s2)  = (max x1 x2, y1+y2)
&gt;                          where (x1,y1) = bbox s1
&gt;                                (x2,y2) = bbox s2
&gt; bbox (LR s1 s2)  = (x1+x2, max y1 y2)
&gt;                          where (x1,y1) = bbox s1
&gt;                                (x2,y2) = bbox s2
```

(b)
In addition to the computations performed by bbox, we have to ensure that the width (height) of two shapes composed by TD (LR) are the same.

```
&gt; rect :: Shape -&gt; Maybe BBox
&gt; rect X          = Just (1,1)
&gt; rect (TD s1 s2) = case (rect s1, rect s2) of
&gt;                     (Just (x1,y1),
&gt;                      Just (x2,y2)) -&gt;
&gt;                        if x1==x2 then Just (x1,y1+y2)
&gt;                                  else Nothing
&gt;                     _                -&gt; Nothing
&gt; rect (LR s1 s2) = case (rect s1, rect s2) of
&gt;                     (Just (x1,y1),
&gt;                      Just (x2,y2)) -&gt;
&gt;                        if y1==y2 then Just (x1+x2,y1)
&gt;                                  else Nothing
&gt;                     _                -&gt; Nothing
```

A very elegant alternative solution, suggested to me by a former student, is to compute the bounding box and compare the number of Xs in the shape to the area of the bounding box.

```
rect :: Shape -&gt; Maybe BBox
rect s | area s == x*y = Just (x,y)
       | otherwise     = Nothing
         where (x,y) = bbox s

area :: Shape -&gt; Int
area X         = 1
area (LR e e') = area e + area e'
area (TD e e') = area e + area e'
```

Even though this implementation is a very clever idea, it violates in a sense the "spirit" of static type checking, because it employs a dynamic semantics, namely the function area, that computes more concrete, fine-grained information than represented by the abstract type BBox.

EXERCISE 3 [4*3+6+6+6 = 30 Points]

(a)

```
&gt; f x y = if null x then [y] else x

&gt; g x  y = if not (null x) then [] else [y]
&gt; g [] y = [y]
```

(1)

```
f :: [a] -&gt; a -&gt; [a]
```

```
g :: [a] -&gt; b -&gt; [b]

(2)
Since null :: [a] -&gt; Bool is applied to x, x must be of type [a].
Assuming that y is of some type b, [y] is of type [b]. Since both
branches of a conditional have to have the same type, the types
of [y] and x have to agree. In other words [b] and [a] have to be
the same. This can be achieved by letting b=a. The result type
of f is determined by the type of the conditional, which is [a].

For g, the same reasoning applies to x and y as in the case for f. The
empty list is of type [c] for any type c. Again, the typing rule for
the conditional forces c and b to be the same (e.g. b) and the result
type to be [b], but b need not be the same as a.

(3)
The type of g is more general since the type of f can be obtained by
substituting a for b. In other words, g works for argument lists of
different types whereas f requires both input lists to be of the
same type.

(4)
Since x is not used in the result of g, its type is unconstrained. In
particular, it is not constrained to match that of [y] as in the
definition of f.


(b)
h :: [b] -&gt; [(a, b)] -&gt; [b]

&gt; h (x:xs) ((y,z):ys) = z:h xs ((y,x):ys)

Much better: h xs [(y,z)] = z:xs


(c)
k :: (a -&gt; b) -&gt; ((a -&gt; b) -&gt; a) -&gt; b

&gt; k f g = f (g f)


(d)
It is difficult to come up with a definition for such a function because
when in a function definition "f x = ..." it is hard to find a value
of type b (since the argument x has type a). So one has to recourse to
tricks like the following.

&gt; f1 x = f1 x

Or:

&gt; f2 x = undefined
&gt;        where undefined = undefined

Or:

&gt; f3 x = head []

Without such tricks, it is impossible to find a definition.
</pre></body></html>
```