<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 [25 Points]

The abstract syntax of the the stack language was already given in
the exercise. The extended abstract syntax is shown below in
exercise 2 (a).

Given the type for a stack, the domain D has to be defined as a
function domain using the Maybe data type to model error values.

> type Stack = [Int]
>
> type StackE = Maybe Stack
>
> type D = StackE -> StackE

A semantic function can always be written in the following form,
mapping from abstract syntax (here, Cmd) into the semantic domain D.

  semCmd :: Cmd -> D

By substituting the definition for the type D, we can obtain

  semCmd :: Cmd -> StackE -> StackE

and by expanding the definition of StackE finally the form shown
below. The semantic function semCmd maps each operation to a
transformation of states, which are given by elements of type
StackE, that is, stacks or errors. Each transformation maps error
states (represented by Nothing) into error states and returns
valid stacks (represented by Just values) only for non-erroneous
input stacks and for commands that find enough arguments.

Note that the "as" pattern "vs@(v:_)" matches the whole list to
vs and at the same time the first element of the list to v.

> semCmd :: Cmd -> Maybe Stack -> Maybe Stack
> semCmd (LD i) (Just s)   = Just (i:s)
> semCmd DUP   (Just vs@(v:_))   = Just (v:vs)
> semCmd ADD   (Just (v1:v2:vs)) = Just (v1+v2:vs)
> semCmd MULT   (Just (v1:v2:vs)) = Just (v1*v2:vs)
> semCmd _   _   = Nothing

Finally, the semantics for programs threads a stack, initialized
to [], through each command of the program.

> sem :: Prog -> Maybe Stack -> Maybe Stack
> sem []   s = s
> sem (c:cs) s = sem cs (semCmd c s)

Here are some test cases:

> tst1 = [LD 3, DUP, ADD, DUP, MULT]
> tst2 = []::Prog
> err1 = [LD 3, ADD]
> err2 = [LD 3, MULT]
> err3 = [DUP]
>
> tests = [tst1,   tst2,   err1,   err2,   err3]
> expect = [Just [36], Just [], Nothing, Nothing, Nothing]
> test = map (\p->sem p (Just [])) tests == expect

EXERCISE 2 [5+10+30 = 45 Points]
</pre>

(a) [5 Points]

The abstract syntax has to be extended by two constructors.

```
&gt; type Prog = [Cmd]
&gt;
&gt; data Cmd = LD Int | ADD | MULT | DUP
&gt;          | DEF String Prog
&gt;          | CALL String
&gt;            deriving Show
```

(b) [10 Points]

The state is given by the stack and the macro definitions. Macro definitions are represented by the following type.

```
&gt; type Macros = [(String,Prog)]
```

There seem to be two possibilities to incorporate error values. Since only operations on the stack can fail, one could be tempted to define:

```
  type State = (Maybe Stack,Macros)
```

However, the purpose of semantic definitions is to assign meanings to programs. With the above definition, the meaning of a program that encounters a stack underflow would be a pair consisting of an error value and a set of macro definitions, whereas the meaning of such a program should be just an error. In other words, in the absence of a dedicated error handling mechanism, programs cannot recover from errors, and thus errors propagate to the top level. Therefore, the following definition is more appropriate.

```
&gt; type State = Maybe (Stack,Macros)
```

(c) [30 Points]

Again the semantic function can be written as mapping from abstract syntax into the semantic domain D, which is given in this case as:

```
  type D = State -&gt; State
```

This gives, in expanded form, the following type for semCmd2.

```
  semCmd2 :: Cmd -&gt; Maybe (Stack,Macros) -&gt; Maybe (Stack,Macros)
```

Arguably more readable is the form using State. The semantics of the stack operation is basically unchanged. The definitions only have to be adapted to the changed semantic domain. Macro definition can be achieved by simply placing a new definition at the beginning of the macro store. Calling a macro means to find it in the store and determine the semantics of the program stored under its name.

```
&gt; semCmd2 :: Cmd -&gt; State -&gt; State
&gt; semCmd2 (LD i)    (Just (s,         m)) = Just (i:s,m)
&gt; semCmd2 DUP       (Just (vs@(v:_),  m)) = Just (v:vs,m)
&gt; semCmd2 ADD       (Just ((v1:v2:vs),m)) = Just (v1+v2:vs,m)
&gt; semCmd2 MULT      (Just ((v1:v2:vs),m)) = Just (v1*v2:vs,m)
&gt; semCmd2 (DEF n p) (Just (s,         m)) = Just (s,(n,p):m)
&gt; semCmd2 (CALL n)  (Just (s,         m)) = case lookup n m of
&gt;                                              Just p -&gt; sem2 p (Just (s,m))
&gt;                                              _      -&gt; Nothing
&gt; semCmd2 _         _                     = Nothing
```

The semantics for programs is defined similar to sem.

```
&gt; sem2 :: Prog -&gt; State -&gt; State
&gt; sem2 []     s = s
&gt; sem2 (c:cs) s = sem2 cs (semCmd2 c s)
```

Here are some test cases:

```
&gt; tests2 = [tst1, tst2, tst3, tst4, tst5, err1, err2, err3, err4, err5, err6]
&gt;  where
&gt;  tst1 = [LD 3, DUP, ADD, DUP, MULT]
&gt;  tst2 = []::Prog
&gt;  tst3 = [DEF "SQR" [DUP,MULT],
&gt;          DEF "DBL" [DUP,ADD],
&gt;          DEF "TRP" [DUP, DUP, ADD, ADD],
&gt;          LD 3, CALL "SQR", CALL "DBL", CALL "TRP"]
&gt;  tst4 = [DEF "SQR" [DUP,MULT],
&gt;          LD 2, CALL "SQR", CALL "SQR", CALL "SQR"]
&gt;  tst5 = [DEF "FOO" [DUP,MULT], -- redef case
&gt;          LD 2,
&gt;          CALL "FOO",
&gt;          CALL "FOO",
&gt;          DEF "FOO" [LD 2, ADD],
&gt;          CALL "FOO"]
&gt;  err1 = [ADD]        -- empty stack cases
&gt;  err2 = [LD 3, ADD]
&gt;  err3 = [MULT]
&gt;  err4 = [LD 3, MULT]
&gt;  err5 = [DUP]
&gt;  err6 = [CALL "FOO"] -- undefined macro
&gt;
&gt; expected2 = [Just [36],Just [],Just [54],Just [256],Just
[18],Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]
&gt; fstM o = case o of {Just (x,_) -&gt; Just x; _ -&gt; Nothing}
&gt; test2 = (map (\p-&gt;fstM (sem2 p (Just ([],[])))) tests2) == expected2
```

EXERCISE 3 [30 Points]

The abstract syntax was given in the exercise as follows. (We use here the names Cmd', State', and sem' to distinguish them from the types/functions defined in earlier exercises so that the file can be loaded into ghci.)

```
&gt; data Cmd' = Pen Mode
&gt;           | MoveTo Int Int
&gt;           | Seq Cmd' Cmd'
&gt;             deriving Show
&gt;
&gt; data Mode = Up | Down
&gt;             deriving Show
```

Likewise, the semantic domain was already given.

```
&gt; type Line   = (Int,Int,Int,Int)
&gt; type Lines  = [Line]
&gt; type State' = (Mode,Int,Int)
```

The semantic function for commands maps a state into a state and a set of lines. The pen commands do not create any lines; they only affect the state of the pen. The meaning of moveto depends on the state of the pen: Only with a pen down will a line be created. In any case, the new position is remembered in the state. For a sequence of two commands, first the set of lines and a new state is determined for the first command, then the effect of the second command is determined using the result state of the first command as input state.

```
&gt; semS :: Cmd' -&gt; State' -&gt; (State',Lines)
&gt; semS (Pen Up)       (_,x,y)    = ((Up,x,y),     [])
&gt; semS (Pen Down)     (_,x,y)    = ((Down,x,y),   [])
&gt; semS (MoveTo x' y') (Up,x,y)   = ((Up,x',y'),   [])
&gt; semS (MoveTo x' y') (Down,x,y) = ((Down,x',y'), [(x,y,x',y')])
&gt; semS (Seq c1 c2)    s          = (s2,           l1++l2)
&gt;                                   where
&gt;                                         (s2,l2) = semS c2 s1
&gt;                                         (s1,l1) = semS c1 s
```

Finally, the semantics function sem calls semS with an initial state and

```
finally, the semantics function semC calls semS with an initial state and
takes from its return value only the second component carrying the list
of drawn lines.

&gt; semC' :: Cmd' -&gt; Lines
&gt; semC' c = snd (semS c (Up,0,0))

</pre></body></html>
```