Arithmetic Expressions

The following `Exp` datatype encodes the abstract syntax for arithmetic expressions. Note that it is recursively defined: `Exp` occurs in the right-hand side of its own definition.

```
data Exp = Const Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
  deriving (Show, Eq)
```

1. Interpreter

- Write an interpreter `eval :: Exp -> Int` that evaluates arithmetic expressions.

Examples

```
Main> eval (Add (Mul (Const 2) (Const 4)) (Const 3))
11
Main> eval (Sub (Const 42) (Mul (Const 6) (Const 7)))
0
```

2. Compiler

Instead of evaluating an arithmetic expression directly, we can also compile it to a program of a simple stack machine and subsequently execute the program. We represent a program as a list of instructions. The instructions `IAdd`, `ISub`, `IMul` take the two topmost elements from the stack, perform the corresponding operation, and push the result onto the stack. The `IPush` instruction pushes the given value onto the stack. A stack is modelled by a list of integers.

```
data Inst = IPush Int | IAdd | ISub | IMul
  deriving (Show, Eq)

type Prog   = [Inst]
type Stack  = [Int]
```

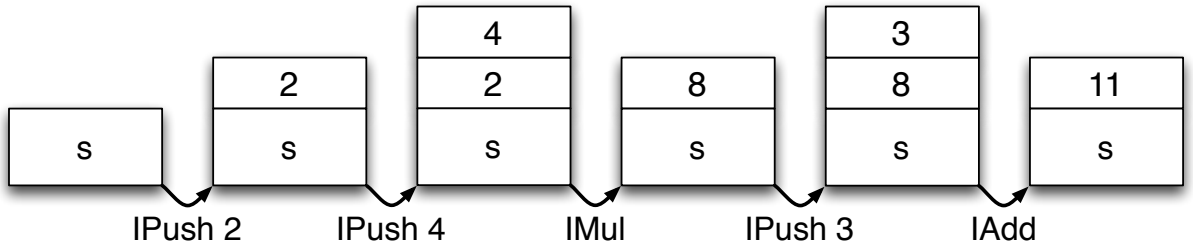
For example, the following arithmetic expression

```
Add (Mul (Const 2) (Const 4)) (Const 3)
```

is equivalent to the stack program

```
[IPush 2,IPush 4,IMul,IPush 3,IAdd]
```

The stack program leaves the result on top of the stack. The stack machine performs the following steps when executing the program on an initial stack `s`



- Write an execution function `execute :: Inst -> Stack -> Stack` that executes a single instruction.

  Since there are cases where the function can crash (e.g. a stack overflow in cases where we want to execute an `IAdd` instruction but the stack contains fewer than two elements), you can use the following exception-raising function where needed:

  ```
  runtimeError :: Stack
  runtimeError = error "Runtime error."
  ```

- Write a function `run :: Prog -> Stack -> Stack` that runs a whole program on a given initial stack.

- Write a compiler `compile :: Exp -> Prog` that compiles arithmetic expressions to stack machine programs. The compiled progam should leave the result of the computation as the top element on the stack. Make sure that your compiler uses a left-to-right evaluation order and that it produces results equivalent to the interpreter, i.e. the following identity holds

  ```
  forall (s :: Stack). run (compile e) s == (eval e) : s
  ```

Examples

```
Main> execute IAdd [4,5,6]
[9,6]

Main> execute ISub [4,5,6]
[1,6]

Main> execute (IPush 2) [4,5,6]
[2,4,5,6]

Main> run [IAdd, ISub] [4,5,6]
[-3]

Main> run [IAdd, ISub, IPush 7, IMul] [4,5,6,8]
[-21,8]
```

```
Main> run [IPush 1,IPush 2,IPush 3,IMul,ISub] []
[-5]

Main> compile (Sub (Const 1) (Mul (Const 2) (Const 3)))
[IPush 1,IPush 2,IPush 3,IMul,ISub]
```