```
<html><head></head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">{-----------
--------------------------------------------------------------------

EECS 662 Spring 2019
Homework 1: Booleans and pairs

This homework assignment starts to explore the foundations of data structures.
In particular, we will study:

 - Booleans, which capture (simple) choice
 - Pairs, which combine multiple values

You will extend the arithmetic language with a set of simple constructs
providing Booleans and pairs.  Because these constructs are suited to specific
kinds of data---you can't add pairs, or deconstruct Booleans---the evaluation
function you build will necessarily be partial.  You will also define an
approximation function which determines what type of data a term produces, as
well as detecting invalid terms.  The challenge problem asks you to extend pairs
to arbitrary-length tuples of values.

-------------------------------------------------------------------------------}

module HW1 where

{-------------------------------------------------------------------------------

As in the previous homework, we'll use the "HUnit" library to support unit
tests.  I have provided a few unit tests along with the assignment.  You may
wish to construct more.

To run the tests, you will need to install the HUnit library.  This should be
doable by issuing a command like

    cabal install HUnit

For how to use HUnit, you can either follow the pattern of my test cases, or
consult the documentation at

   http://hackage.haskell.org/package/HUnit

-------------------------------------------------------------------------------}

import Control.Monad (guard)
import Test.HUnit

------------------------------------------------
-- Dragon eggs (dragons hatch later in the file)
import Text.Parsec hiding (parse)
import Text.Parsec.Language
import Text.Parsec.Token as T
------------------------------------------------

{--------------------------------------------------------------------------------

Term structure
==============

We extend our base arithmetic language (IConst, Plus, Times) with several new
language constructs:

 - For Booleans, we introduce Boolean constants, several Boolean operations
   (conjunction and not), an "is zero" test for integers, and an "if" expression.

 - For pairs, we introduce pair terms "(1, True)" to create pairs, and first and
   second projections ("fst" and "snd") to extract values from pairs.


For example, a term in our extended language might look something like:

    if isz (4 + 5)
    then (isz 4, isz 5)
    else (False, False)

which would be represented by the Haskell value

    If (IsZ (Plus (Const 4) (Const 5)))
       (Pair (IsZ (Const 4)) (IsZ (Const 5)))
       (Pair (BConst False) (BConst False))

Or, we might have a term like

    (fst (1, 2), snd (1, 2))

which would be represented in Haskell as:

    Pair (Fst (Pair (IConst 1, IConst 2)))
         (Snd (Pair (IConst 1, IConst 2)))

--------------------------------------------------------------------------------}

data Term = IConst Int | Plus Term Term | Times Term Term | IsZ Term
          | BConst Bool | And Term Term | Not Term | If Term Term Term
          | Pair Term Term | Fst Term | Snd Term
  deriving (Eq, Show)

{--------------------------------------------------------------------------------

Problem 1: Evaluation (regular)
===============================

Our first task is to implement an evaluation relation for our extended language.
For the most part, the evaluation rules should follow your intuition about the
language features in question.  Writing out the non-trivial rules formally, we
have:


    t ⇓ 0         t ⇓ n
    ------------  ------------- (n ≠ 0)
    isz t ⇓ True  isz t ⇓ False

    t1 ⇓ b1
    t2 ⇓ b2                t ⇓ b
    -------------------    ----------
    t1 &amp;&amp; t2 ⇓ b1 ∧ b2    not t ⇓ ¬b

    t1 ⇓ True                    t1 ⇓ False
    t2 ⇓ v                       t3 ⇓ v
    -------------------------    -------------------------
    if t1 then t2 else t3 ⇓ v    if t1 then t2 else t3 ⇓ v

    t1 ⇓ v1
    t2 ⇓ v2                t ⇓ (v1, v2)   t ⇓ (v1, v2)
    -------------------    ------------   ------------
    (t1, t2) ⇓ (v1, v2)    fst t ⇓ v1     snd t ⇓ v2

However, these (apparently) simple rules hide two new technical challenges.
First, we have previously defined evaluation as a function from terms to
integers (values).  Now, our notion of "value" needs to become more nuanced:
evaluation may produce integers, Booleans, or pairs.  Second, there are plenty
of cases in which our evaluation function is not defined, such as "fst True" or
"if 1 then 2 else 3".  Our implementation function will have to represent these
cases somehow.

--------------------------------------------------------------------------------}

{--------------------------------------------------------------------------------

Our first problem is representing values.  We'll do this by introducing a new
type, called "Value", which captures the three different kindsofvalues:

 - VInt: integer values
 - VBool: Boolean values
 - VBool: Boolean values
 - VPair: pairs of values

As with the other definitions we've given in this course, you could view this as
a mathematical specification of a set V as the smallest set such that:

 - If z is an integer, then z ∈ V
 - If b is a Boolean constant, then b ∈ V
 - If v₁ ∈ V and v₂ ∈ V, then (v₁, v₂) ∈ V

Alternatively, you can think of them as terms that don't compute any further,
but also don't correspond to errors.

-------------------------------------------------------------------------------}

data Value = VInt Int | VBool Bool | VPair Value Value
  deriving (Eq, Show)



{-------------------------------------------------------------------------------

With a definition of values in hand, we can go on to define evaluation.  To
account for the undefined cases of evaluation, we'll use Haskell's standard
`Maybe` type.  Recall that `Maybe` is defined as

    data Maybe a = Nothing | Just a

so your evaluation function should return `Nothing` for terms that don't have a
well-defined meaning (such as "fst True" or "if 1 then 2 else 3"), and should
return `Just v` if the meaning of a term is the value `v`.

You may want to define some helper functions for common cases of manipulating
`Maybe` types, as we did when implementing evaluators in class.

-------------------------------------------------------------------------------}

eval :: Term -&gt; Maybe Value
eval = error "Not defined"

evalTests = test [eval (parse s) ~?= Just v | (s, v) &lt;- tests]
    where tests = [ ("if isz 1 then 2 else 3", VInt 3)
                  , ("fst (1, True) + snd (True, 2)", VInt 3)
                  , ("isz 0 &amp;&amp; not (isz 1)", VBool True) ]

{--------------------------------------------------------------------------------

Part 2: Typing (regular)
========================

Similar to our "safety" property for the arithmetic language, which guaranteed
when terms would evaluate fully (i.e., not attempt to divide by zero), we will
property for the extended language that excludes terms that can't evaluate.
This property, called "typing", will combine aspects of the approximation and
safety relations for the arithmetic language: in addition to guaranteeing that
terms evaluate, it will characterize what type of values they produce.


--------------------------------------------------------------------------------}

{-------------------------------------------------------------------------------

We start out by introducing a definition of types.  Following the approximation
idea, types approximate possible values: integer values (VInt z) are
approximated by the Int type, Boolean values are approximated by the Bool type,
and so forth.

-------------------------------------------------------------------------------}

data Type = TInt | TBool | TPair Type Type
  deriving (Eq, Show)
```

```
{--------------------------------------------------------------------------------

Next, we define a function which computes the type of a given term.  We write
the relation between terms (t) and their types (T) with a colon (t : T); so, we
might have "1 : Int" or "isz 1 : Bool".  As with evaluation, we can
characterizing typing using inference rules.  Some representative inference
rules for our language and types:

                          t₁ : Int
                          t₂ : Int
    -------- (z ∈ ℤ)     -------------
    z : Int               t₁ + t₂ : Int

    t₁ : T₁
    t₂ : T₂               t : (T₁, T₂)
    -------------------   ------------
    (t₁, t₂) : (T₁, T₂)   fst t : T₁

    t₁ : Bool
    t₂ : T
    t₃ : T
    -------------------------
    if t₁ then t₂ else t₃ : T

As with the evaluation relation, our typing relation is partial.  There is no
type for terms like "fst True" or "if 1 then 2 else 3", because those terms do
not evaluate to values.  Correspondingly, our typing function produces a `Maybe
Type`, returning `Nothing` for terms without types.

Your task is to complete the implementation of the typing function `check`.  You
should be able to deduce the typing rules for the remaining terms... follow the
requirements of your evaluation function.

--------------------------------------------------------------------------------}

check :: Term -&gt; Maybe Type
check = error "Not implemented"


checkTests = test $ [check (parse s) ~?= Just t | (s, t) &lt;- successes] ++
                    [check (parse s) ~?= Nothing | s &lt;- failures]
    where successes = [ ("if isz 1 then 2 else 3", TInt)
                      , ("fst (1, True) + snd (True, 2)", TInt)
                      , ("isz 0 &amp;&amp; not (isz 1)", TBool) ]
          failures  = [ "if 1 then 2 else 3"
                      , "fst True"
                      , "(1, 2) + 3" ]





{---------------------------------------------------------------------------------

Part 3: n-ary Products (challenge)
==================================

A common technique in studying programming languages is to translate complicated
language features to simpler "core" language features.  This allows us study
practical languages while still using simple formal approaches, as we will be
using in this class.

This problem asks you to do such a translation.  We would like to support
arbitrary-size tuples in our source code... so, instead of just having pairs
(t₁, t₂), we can have (t₁, t₂, .. tᵢ).  Similarly, we need to be able to access
any component of a tuple, so instead of just having "fst t" and "snd t", we have
"prj[i] t" for any i ≥ 0.

To interpret the arbitrary sized tuples, we will convert them into nested pairs.
We do this with two functions:
```

 - npair takes a list of terms (t_1, t_2, ..., t_i) and returns a single nested
   pair containing all of its arguments.

 - prnj takes a term constructed by npair and an index i and returns the ith
   term of the original list.

So, for example, you should have:

```
    eval (prjn 0 (npair [Const 1, Const 2, Const 3])) == VInt 1
```

and

```
    eval (prjn 2 (npair [Const 1, Const 2, Const 3])) == VInt 3
```

Your task is to implement these two functions. Note that you do not know when
interpreting `prjn i` how many values there were in the original pair, so your
translation should work for any pair that includes i elements. This should
force the structure of the term returned by `npair`.

The parser knows about npair and prjn.. so if you say:

```
    parse "(1, 2)"
```

you will get Pair (IConst 1) (IConst 2), but if you say

```
    parse "(1, 2, 3)"
```

the parser will call `npair [Const 1, Const 2, Const 3]`.

```
--------------------------------------------------------------------------------}

npair :: [Term] -&gt; Term
npair = error "Not implemented"

prjn  :: Int -&gt; Term -&gt; Term
prjn = error "Not implemented"

naryTests = test [eval (parse s) ~?= Just v | (s, v) &lt;- tests]
    where tests = [ ("prj[1] (1, 2, 3) + prj[2] (1, 2, 3)", VInt 5)
                  , ("prj[0] (True, 1, (True, 1, 2)) &amp;&amp; prj[0] (prj[2] (True, 1, (True,
1, 2)))", VBool True) ]


{--------------------------------------------------------------------------------


Here be dragons

--------------------------------------------------------------------------------}

parse :: String -&gt; Term
parse s = case runParser (terminal exprp) () "" s of
          Left err -&gt; error (show err)
          Right t  -&gt; t
    where l = makeTokenParser $
              haskellDef { reservedNames = ["True", "False", "if", "then", "else", "isz", "not",
"fst", "snd", "prj"]
                         , reservedOpNames = ["+", "*", "&amp;&amp;"] }

          terminal p = do x &lt;- p
                          eof
                          return x
          identifier = T.identifier l
          reserved = T.reserved l
          reservedOp = T.reservedOp l
          parens = T.parens l
          brackets = T.brackets l
          lexeme = T.lexeme l
          comma = T.comma l
          commaSep1 = T.commaSep1 l
```

```
        exprp = condp

        condp = choice [ do reserved "if"
                            cond &lt;- exprp
                            reserved "then"
                            cons &lt;- exprp
                            reserved "else"
                            alt &lt;- exprp
                            return (If cond cons alt)
                       , addp ]
        addp = chainl1 multp (choice [ reservedOp "+" &gt;&gt; return Plus
                                     , reservedOp "&amp;&amp;" &gt;&gt; return And ])
        multp = chainl1 atomp (reservedOp "*" &gt;&gt; return Times)

        atomp = choice [ IConst `fmap` lexeme intConst
                       , BConst `fmap` lexeme boolConst
                       , do unaop &lt;- unary
                            e &lt;- atomp
                            return (unaop e)
                       , do reserved "prj"
                            n &lt;- brackets intConst
                            guard (n &gt;= 0)
                            e &lt;- atomp
                            return (prjn n e)
                       , do es &lt;- parens (commaSep1 exprp)
                            case es of
                              [e]      -&gt; return e
                              [e1, e2] -&gt; return (Pair e1 e2)
                              _        -&gt; return (npair es) ]

        intConst = do ds &lt;- many1 digit
                      return (read ds)

        boolConst = choice [ reserved "True" &gt;&gt; return True
                           , reserved "False" &gt;&gt; return False ]

        unary = choice [ reserved "isz" &gt;&gt; return IsZ
                       , reserved "not" &gt;&gt; return Not
                       , reserved "fst" &gt;&gt; return Fst
                       , reserved "snd" &gt;&gt; return Snd
                       ]
</pre></body></html>
```