2009 Haskell January Test
Binary Decision Diagrams

This test comprises four parts and the maximum mark is 25. Parts I, II and III are worth 23 of the 25 marks available. The **2009 Haskell Programming Prize** will be awarded for the best attempt(s) at Part IV.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You may therefore wish to define your own tests to complement the ones provided.

1 Introduction

Binary Decision Diagrams, or BDDs, are directed acyclic graphs that are used to represent boolean expressions[1]. Operations on BDDs are often substantially more efficient than the equivalent operations on expressions and this makes them attractive when working with large-scale problems involving boolean logic. Common applications of BDDs include circuit synthesis and "Satisfiability Solvers" (SAT solvers) that are used extensively in verification.

2 Boolean Expressions

In this exercise you are going to manipulate an abstract representation of well-formed boolean expressions that comprise the boolean constants $False$ and $True$, boolean variables (e.g. x_1, x_2,...), negation ($\neg$), and the binary operators *and* ($\wedge$) and *or* ($\vee$). Here are some examples:

$$
\begin{array}{l}
False \\
x_1 \wedge True \\
\neg(x_7 \wedge (x_2 \vee \neg x_3)) \\
\neg x_3 \vee (x_2 \vee \neg x_9)
\end{array}
$$

The operations $\neg$, $\wedge$ and $\vee$ have the usual meaning, as shown in Table 1. Boolean expressions

a	$\neg a$
$False$	$True$
$True$	$False$

a	b	$a \wedge b$	$a \vee b$
$False$	$False$	$False$	$False$
$False$	$True$	$False$	$True$
$True$	$False$	$False$	$True$
$True$	$True$	$True$	$True$

Table 1: Truth tables for $\neg$, $\wedge$ and $\vee$.

such as these can be represented in Haskell using the following algebraic data type:

```
type Index = Int

data BExp = Prim Bool | IdRef Index | Not BExp | And BExp BExp | Or BExp BExp
          deriving (Eq, Ord, Show)

type Env = [(Index, Bool)]
```

with `Not`, `And` and `Or` representing $\neg$, $\wedge$ and $\vee$ respectively. Note that boolean variables are represented by a *positive* integer that uniquely identifies the variable, e.g. 5 for x_5. As an example, the boolean expression $\neg(x_1 \wedge (False \vee x_2))$ will be represented thus:

```
Not (And (IdRef 1) (Or (Prim False) (IdRef 2)))
```

[1]Here, boolean expressions are synonymous with boolean *functions* in the sense that an expression defines a mapping from boolean variable values to one of $\{False, True\}$.

3 Binary Decision Diagrams

An alternative way to represent a boolean expression is as a *Binary Decision Diagram*, or BDD, which provides a succinct and often very efficient way to implement boolean operations. Each internal node in a BDD specifies an integer variable index, e.g. 5 for x_5. This variable can assume either the value $False$ or $True$ and there is a separate subtree for each case; we will adopt the convention that the left subtree always corresponds to $False$ and the right subtree to $True$. Each path through the tree thus represents a unique assignment of values to variables and leads to a *leaf* that represents the value of the expression under the corresponding variable assignment; again, this is either $False$ or $True$.

A *complete* BDD is conceptually very similar to a binary tree that encodes all possible assignments of the boolean variables contained in a given expression. If there are n such variables, then there are 2^n possible assignments. For the bulk of this exercise BDDs will be assumed to be complete in this sense; such BDDs are sometimes referred to as *Binary Decision Trees*.

As an example, Figure 1 shows two BDDs for the expression $\neg(x_1 \wedge (False \vee x_2))$. The leaves F and T in the figure are shorthand for $False$ and $True$ respectively. The number associated with

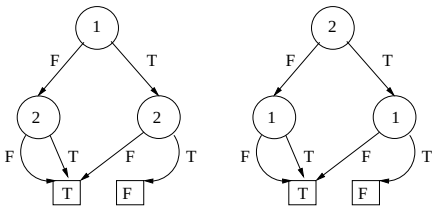

Figure 1: The two complete BDDs for $\neg(x_1 \wedge (False \vee x_2))$

each internal node is the variable index. Each variable in an expression corresponds to one layer in the tree, so there are only two possible BDDs in this case: one where the variable references appear in the order [1,2] from top to bottom (left) and the other where they appear in the order [2,1] (right).

As an example, when $x_1 = True$ and $x_2 = False$, the expression evaluates to $True$ because the corresponding path through the BDD ($right, left$ for the BDD on the left of Figure 1 and $left, right$ for the one on the right) leads to the leaf corresponding to $True$ (T).

Notice that there is only one leaf corresponding to the expression $False$ (F), similarly $True$ (T), and that these are *shared* by the internal nodes. Indeed, we shall see examples later on where the internal nodes of a BDD can also be shared. BDDs in general are thus *Directed Acyclic Graphs* (DAGs) and can be represented in Haskell by a pair comprising the identifier of the root node of the BDD and a list of internal nodes that collectively describe the structure of the BDD. Each internal node comprises a unique node identifier, a *positive* integer, and a triple (i, l, r), where i is the integer index of a boolean variable and l and r are the identifiers of the nodes corresponding to the left and right subtrees respectively:

```
type NodeId = Int

type BDDNode =  (NodeId, (Index, NodeId, NodeId))

type BDD = (NodeId, [BDDNode])
```

Because the leaves representing $False$ and $True$ are potentially part of every BDD, they will not be explicitly included in the representation. Instead, the reserved labels 0 and 1 will be used to

label the 'invisible', but ever-present, leaves corresponding to $False$ and $True$ respectively. For example, the BDDs in Figure 1 might be represented respectively as:

```
(2,[(4,(2,1,1)),(5,(2,1,0)),(2,(1,4,5))])
(2,[(2,(2,7,3)),(7,(1,1,1)),(3,(1,1,0))])
```

Note that the node identifiers must be unique but are otherwise arbitrary and that the order of the elements in the list is not significant. What if there are no internal nodes, as in the case of the expression $False$, for example? The corresponding BDD in this case contains no nodes other than that for $False$ itself and so will be represented by the pair `(0,[])`.

3.1 Operations on BDDs

One of the advantages of working with BDDs is that various operations on boolean expressions can be performed directly on the corresponding BDD, often much more efficiently than if they were performed on the expressions themselves. In this exercise you will implement two such operations:

1. *checkSat* (check satisfiability) which checks whether a given set of variable assignments *satisfies* the corresponding expression, i.e. causes it to be $True$. For example, in the BDD of Figure 1, the assignment $(x_1 = True, x_2 = False)$ satisfies the expression but $(x_1 = True, x_2 = True)$ does not. To check for satisfiability in a BDD you simply traverse the BDD from top to bottom proceeding either left or right depending on the given assignment ($False \Rightarrow$ turn left, $True \Rightarrow$ turn right). The expression is satisfied iff you end up at the 'invisible' leaf corresponding to $True$ (node identifier 1).

2. *allSat*, which returns the set of all variable assignments that will cause the expression to be $True$. For example, for the BDD of Figure 1 the 'satisfying' variable assignments are $(x_1 = False, x_2 = False)$, $(x_1 = False, x_2 = True)$ and $(x_1 = True, x_2 = False)$. Each element of the set corresponds to a path from the root node of the BDD that ends with the leaf corresponding to $True$ (node identifier 1).

4 What to do

The exercise is broken down into four parts. The majority of the marks are assigned to the first three parts. You are advised not attempt the last part until you have completed the first three.

A number of example expressions, `b1`, ..., `b8`, each of type `BExp`, are included in the template file for testing purposes, including those referred to in the text:

`b1`	$False$
`b2`	$\neg(x_1 \wedge (False \vee x_2))$
`b3`	$x_1 \wedge True$
`b4`	$x_7 \wedge (x_2 \vee \neg x_3)$
`b5`	$\neg(x_7 \wedge (x_2 \vee \neg x_3))$
`b6`	$(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$
`b7`	$\neg x_3 \vee (x_2 \vee \neg x_9)$
`b8`	$x_1 \vee \neg x_1$

Note that `b8` is a *tautology*: the expression is $True$ regardless of the value of x_1. A valid BDD for each expression, assuming the indices are listed in ascending order (e.g. `[2,3,7]` for `b4`), is also included in the template. These are labelled `bdd1`, ..., `bdd8`.

Part I - Basics

1. Define a polymorphic function `lookUp :: Eq a => a -> [(a, b)] -> b` that will perform generic table look-up. A precondition is that there is exactly one occurrence of the item being searched (type `a`) in the table. For example, `lookUp 5 [(4,1),(5,2),(9,9)]` should return 2 and `lookUp (2,1) [((1,2),True),((2,1),False)]` should return `False`.

   [1 mark]

2. Define a function `checkSat :: BDD -> Env -> Bool` that will return `True` if the variable assignments given in the environment `Env` satisfy the expression corresponding to the given BDD; `False` otherwise. An *environment* (type `Env`) maps a variable index to its value, which will be either `False` or `True`. The type synonym `Env` is defined in the template as follows:

   ```
   type Env = [(Index, Bool)]
   ```

   For example,

   ```
   *Main> checkSat bdd2 [(1,True),(2,False)]
   True
   *Main> checkSat bdd7 [(3,True),(2,False),(9,True)]
   False
   ```

   Hint: You'll need a helper function to walk you from top to bottom through the BDD, starting with the root node. You'll need *two* table look-ups here: one to look up the current node identifier in the BDD and the other to look up the variable index (given in the node) in the given environment. Don't forget that you can assume nothing about the internal node identifiers in the BDD, except that they are positive and unique, nor the order of the elements in either the BDD or the environment.

   [4 marks]

3. Define a function `sat :: BDD -> [[(Index, Bool)]]` that will compute the list of all sets of variable assignments that cause the boolean expression corresponding to the given BDD to be $True$. Each 'set' of assignments is a list of `(Index, Bool)` pairs. For example,

   ```
   *Main> sat bdd1
   []
   *Main> sat bdd2
   [[(1,False),(2,False)],[(1,False),(2,True)],[(1,True),(2,False)]]
   *Main> sat bdd8
   [[(1,False)],[(1,True)]]
   ```

   Hint: You'll need a helper function similar to that for `checkSat` above, wherein the base cases should be `[]` when you hit the node representing $False$ (identifier 0) and `[[]]` when you hit the node representing $True$. This is because there is no satisfying assignment for the expression $False$ and exactly *one* satisfying assignment for $True$ – the empty one.

   [5 marks]

Part II

1. Define a *non-recursive* function `simplify :: BExp -> BExp` that will simplify a given boolean expression by applying the rules defined in the truth tables of Table 1 in the cases where the argument(s) to $\neg$, $\wedge$ and $\vee$ are primitive booleans (`Prim`). If they are not then the expression should be returned unmodified. For example,

```
Main> simplify (Not (Prim False))
Prim True
Main> simplify (Or (Prim False) (Prim False))
Prim False
Main> simplify (And (IdRef 3) (Prim True))
And (IdRef 3) (Prim True)
```

   Note that the third example could be further simplified to `IdRef 3` by exploiting the semantics of $\wedge$, but this type of simplification is not required here.

   [2 marks]

2. Define a recursive function `restrict :: BExp -> Index -> Bool -> BExp` that replaces a specified variable in a boolean expression, given by its integer `Index`, with a given boolean constant. Note that the replacement may allow one or more of the simplification rules above to be applied, so you should invoke your `simplify` function each time you build a new `Not`, `And` or `Or` expression. For example,

```
*Main> b7
Or (Not (IdRef 3)) (Or (IdRef 2) (Not (IdRef 9)))
*Main> restrict b7 2 True
Or (Not (IdRef 3)) (Or (Prim True) (Not (IdRef 9)))
*Main> restrict (restrict b7 2 True) 9 False
Or (Not (IdRef 3)) (Prim True)
*Main>  restrict (restrict (restrict b7 2 True) 9 False) 3 True
Prim True
```

   [5 marks]

Part III

You are now going to build the BDD for a given expression by defining a function

```
buildBDD :: BExp -> [Index] -> BDD
```

The `[Index]` contains the variable indices referenced in the expression, in some order. A precondition is that this contains exactly one occurrence of each identifier in the expression and no other elements.

You can implement this function in any way you wish, but the suggested method is in terms of a helper function whose declaration is included in the template file:

```
buildBDD' :: BExp -> NodeId -> [Index] -> BDD
```

The `NodeId` is the identifier that you should assign to the next node in the BDD, as you build it. Because the node identifiers 0 and 1 have been reserved to represent $False$ and $True$ respectively, the first unused identifier is 2. From this, it is easy to see how `buildBDD` should be defined (as can be found in the template):

```
buildBDD e xs
  = buildBDD' e 2 xs
```

Figure 2 illustrates the idea. The numbers to the left of each node are the node identifiers – see below for how to assign them. In this example, the `Index` list passed to `buildBDD` would have

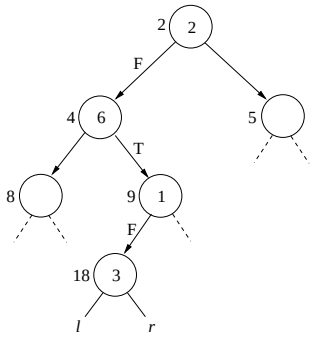

Figure 2: Building a BDD

been of the form `[2,6,1,3,`...*etc.*`]`. The diagram shows the situation where the node labelled 18 is about to be built. The corresponding call to `buildBDD'` would be of the form:

```
buildBDD' e 18 [3,...]
```

In this case the next node will be of the form `(18,(3,l,r))`, where `l` and `r` are the integer identifiers associated with the left and right 'subtrees'. You will obtain these by recursion. Note that when you recurse, you will need to *restrict* the current expression, `e`, with either `(3,False)` or `(3,True)` depending on whether you are recursing into the left or right subtree respectively. Use your `restrict` function above to do this.

Remark: This use of restriction is the essence of the *Shannon Expansion*, which says that a boolean expression E can be rewritten to $x \rightarrow E[True/x], E[False/x]$, where $E[b/x]$ means E with x replaced throughout by b (restriction) and where $P \rightarrow Q, R$ means "if P then Q else R".

Choosing Node Identifiers

As you build the BDD you need to ensure that each node is given a unique identifier. The easiest way to do this is for a node labelled n to arrange for the root nodes of its left and right subtrees to be labelled $2n$ and $2n + 1$ respectively, but see below for the base case. You'll see that this gives you a unique labelling in any binary structure. Figure 2 illustrates this. Two additional nodes (labels 5 and 8) have been included in the diagram to make the numbering scheme clear.

When the `Index` list is empty (`[]`) you have reached a leaf in the tree. In this case you return a pair of the form `(b,[])` where `b` is either 0 or 1 depending on the value of the expression `e`. Crucially, if you have implemented `restrict` correctly then the expression will be either `Prim False` or `Prim True` because you will by now have given each variable in the expression a value.

[6 marks]

Part IV

You are advised not to attempt this part until you have completed Parts I-III above. It is worth only 2 of the 25 marks available.

Reduced Ordered BDDs

It usually turns out that a complete BDD contains a lot of redundancy. If this is removed then it is often possible to reduce substantially the amount of memory required to store the BDD. The last part of this exercise invites you to build these more optimal representations, referred to as *Reduced Ordered BDDs* or ROBDDs.

There are two optimisations that you can perform on a BDD to make an ROBDD. The first eliminates a node if the two subtrees it refers to are the same, as illustrated in Figure 3. The

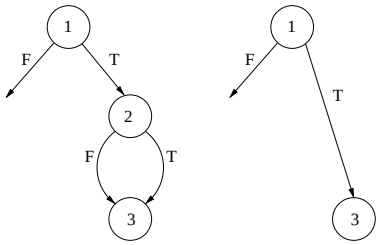

Figure 3: Node elimination

second shares subtrees wherever possible. An example is shown in Figure 4. Here, the subtree rooted at X is identical to the subtree rooted at Y. If node X already exists at the point where node Y is about to be built, the whole subtree that would otherwise be generated can be avoided, and instead we can generate a reference directly to X. If, for example, that subtree were being built as a consequence of building some other node, Z say (as in the diagram), then the effect of the optimisation will be for Z to be linked directly to X, as shown.

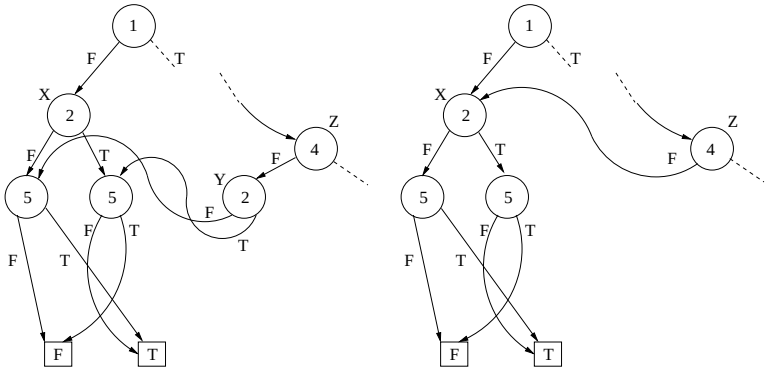

Figure 4: Subtree sharing

To illustrate the effect of the above optimisations, first see Figure 5 which shows the complete (unoptimised) BDD for the expression

$$(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$$

This corresponds to `b6` in the template. Notice the enormous amount of redundancy.

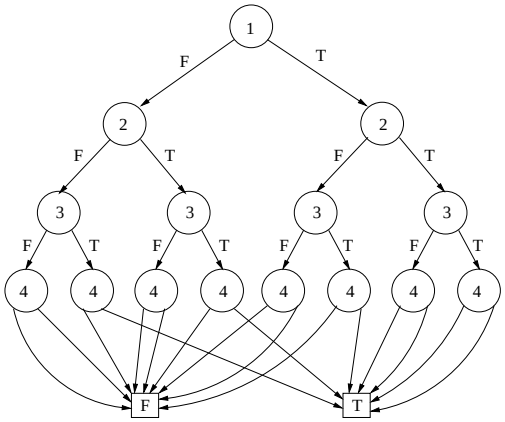

Figure 5: The unoptimised BDD for $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

Now look at Figure 6 which shows two possible ROBDDs for the same expression. Notice that they contain many fewer nodes than the unoptimised case. Interestingly, the structure of the resulting ROBDD depends on the order in which the variable indices are listed during the build. Indeed, the effect on both size and structure can sometimes be quite radical. The ROBDDs of Figure 6 were obtained using the variable index order `[1,3,2,4]` and `[1,2,3,4]` respectively.[2] Thus, define a function `buildROBDD :: BExp -> [Index] -> ROBDD` that builds an ROBDD by

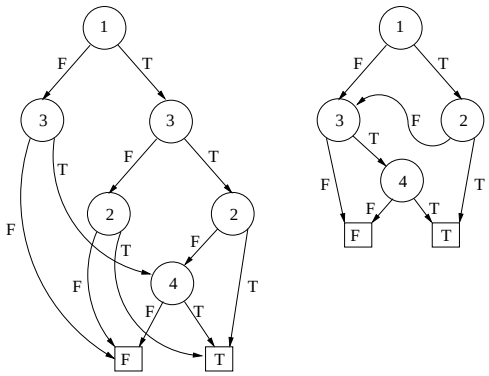

Figure 6: Two ROBDDs for $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

applying the above optimisations as the ROBDD is being built.

[2 marks]

[2] Determining the variable order that results in the smallest ROBDD is an example of an *NP-complete* problem; in practice, this means that you have to try all possible variable orderings in order to find the best one, which is computationally intractable for large n.