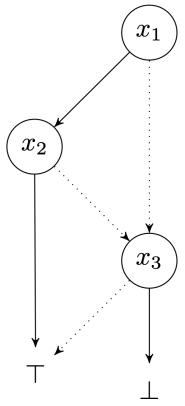
Challenge 2

Binary Decision Diagrams

Background

Binary Decision Diagrams (BDD) allow extremely compact representations of propositional formulas by sharing common sub-expressions. Formulas are stored as directed acyclic graphs, where each internal node represents the conditional test of a variable x. Two leaf nodes are used to indicate the values $\top$ and $\bot$. Using a BDD, the formula $(x_1 \wedge x_2) \vee x_3$ would be represented as the following graph:



Dotted edges are used to indicate paths where the variable is false, while solid edges indicate paths where the variable is true.
A Reduced Ordered BDD (ROBDD) has the following additional properties:

1. A BDD is **ordered** if its variables appear in the same order along every path
2. A BDD is **reduced** if neither of the following rules can be applied:
    a. Eliminate any nodes whose children are isomorphic
    b. Combine isomorphic subgraphs

We can implement a simple library to build ROBDDs in Ocaml in the following manner:

```
Unset
type var = int
type node =
  | Node of { var: int; left: node; right: node; }
  | True
  | False

let equal n m = match n, m with
  | True, True -> true
  | False, False -> true
  | Node n, Node m -> n.var = m.var && n.right == m.right && n.left == m.left
```

```
  | _ -> false

module Tbl = Hashtbl.Make(struct
  type t = node
  let equal = equal
  let hash = Hashtbl.hash
end)

type bdd = node Tbl.t
let mk_bdd () : bdd = Tbl.create 32

let mk_node (b : bdd) (n : node) : node =
  match Tbl.find_opt b n with
  | Some n -> n
  | None -> Tbl.add b n n; n

let mk_true b : node = mk_node b True
let mk_false b : node = mk_node b False

let mk_if b var left right =
  if equal left right then left else
  mk_node b (Node { var; left; right })

let mk_var b v = mk_if b v (mk_true b) (mk_false b)

let rec mk_not (b : bdd) (n : node) : node = match n with
  | True -> mk_false b
  | False -> mk_true b
  | Node { var; left; right } -> mk_if b var (mk_not b left) (mk_not b right)

let rec mk_and (b : bdd) (l : node) (r : node) : node = match l, r with
  | True, _ -> r
  | _, True -> l
  | False, _ | _, False -> mk_false b
  | Node {var = vara; left = lefta; right = righta }, Node { var = varb; left =
leftb; right = rightb } ->
  begin match compare vara varb  with
  | -1 -> mk_if b vara (mk_and b lefta r) (mk_and b righta r)
  | 0 -> mk_if b vara (mk_and b lefta leftb) (mk_and b righta rightb)
  | 1 -> mk_if b varb (mk_and b l leftb) (mk_and b l rightb)
  | _ -> assert false
  end
```

In this implementation each BDD node holds references to its two children, forming the DAG of a formula.

Tasks

Translate the BDD library above into the language of your choice and use a verification tool to check the following properties:

1. Verify the memory safety
2. Verify the termination
3. Verify the operations are correct:
    a. Applying `and(a, b)` should produce a node equivalent to the conjunction of the formulas represented by `a` and `b`
    b. Applying `not(a)` should produce a node equivalent to the negation of the formulas represented by `a`
4. Verify that each operation preserves the properties of a ROBDD
    a. If the input BDD is ordered, then the output of each operation is an ordered BDD
    b. Similarly, if the input BDD is reduced, then the output is also reduced

Appendix

Here is an implementation in Java and one in C. Use them, adapt them, or ignore them as you wish. They have been tested some but absence of defects is not guaranteed!

Java

```java
import java.util.LinkedHashMap;
import java.util.HashMap;
import java.io.PrintStream;

/** A BDD Node */
class Node {
    int id = -1;
    boolean isTrue() {
        return this instanceof LeafNode && ((LeafNode)this).val;
    }
    boolean isFalse() {
        return this instanceof LeafNode && !((LeafNode)this).val;
    }
}

/** A leaf node: either "true" or "false. */
class LeafNode extends Node {
    boolean val;
    public LeafNode(boolean val) {
```

```
        this.val = val;
    }
    public String toString() {
        return "Node["+id+";"+val+"]";
    }
    public boolean equals(Object obj) {
        return obj instanceof LeafNode && val == ((LeafNode)obj).val;
    }
    public int hashCode() {
        return val ? 1 : 0;
    }
}

/* A non-leaf node: has a variable and two children */
class CompoundNode extends Node {
    int var;
    Node low; // where to go if var is false
    Node high; // where to go if var is true
    CompoundNode(int var, Node low, Node high) {
        this.var = var;
        this.low = low;
        this.high = high;
    }
    public String toString() {
        return "Node["+id+";"+var+","+low.id+","+high.id+"]";
    }
    public boolean equals(Object obj) {
        if (obj instanceof CompoundNode) {
            CompoundNode that = (CompoundNode)obj;
            return var==that.var && low==that.low && high==that.high;
        }
        return false;
    }
    public int hashCode() {
        return (1<<24) + 37*var + (1<<10)*23*low.hashCode()
            + 19*high.hashCode();
    }
}

public class BDD {
    final static PrintStream out = System.out;
    HashMap<Node, Node> table = new LinkedHashMap<>();
    Node getNode(Node node) {
        if (node instanceof CompoundNode) {
            CompoundNode that = (CompoundNode)node;
            if (that.low == that.high) return that.low;
        }
```

```
        Node result = table.get(node);
        if (result == null) {
            node.id = table.size();
            table.put(node, node);
            return node;
        }
        return result;
    }
    public void print() {
        for (Node node : table.keySet())
            out.println(node);
    }
    public Node node(boolean val) {
        return getNode(new LeafNode(val));
    }
    public Node node(int var, Node low, Node high) {
        return getNode(new CompoundNode(var, low, high));
    }
    public Node not(Node node) {
        if (node.isTrue())
            return node(false);
        if (node.isFalse())
            return node(true);
        CompoundNode u = (CompoundNode)node;
        return node(u.var, not(u.low), not(u.high));
    }
    public Node and(Node node1, Node node2) {
        if (node1.isTrue())
            return node2;
        if (node2.isTrue())
            return node1;
        if (node1.isFalse() || node2.isFalse())
            return node(false);
        CompoundNode u = (CompoundNode)node1, v = (CompoundNode)node2;
        if (u.var < v.var)
            return node(u.var, and(u.low, v), and(u.high, v));
        else if (u.var == v.var)
            return node(u.var, and(u.low, v.low), and(u.high, v.high));
        else
            return node(v.var, and(u, v.low), and(u, v.high));
    }

    public static void main(String[] args) {
        BDD b = new BDD();
        Node f = b.node(false), t = b.node(true), x = b.node(1, f, t);
        assert b.and(x, b.not(x)) == f;
    }
```

```
}
```

C/C++

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>

#define LEN 100 // hash table length

typedef struct Node {
  enum NodeKind { LEAF, COMPOUND } kind;
  int id;
  struct Node * nxt; // for hashtable chaining
  union {
    struct LeafPart {
      bool val;
    } leaf;
    struct CompoundPart {
      int var;
      struct Node * low;
      struct Node * high;
    } compound;
  };
} Node;

int size = 0;
Node * bdd[LEN]; // hashtable of nodes

void node_print(Node * u) {
  printf("Node[%d;", u->id);
  if (u->kind == LEAF)
    printf("%s]", u->leaf.val ? "true" : "false");
  else
    printf("%d,%d,%d]", u->compound.var, u->compound.low->id,
         u->compound.high->id);
}

void print(void) {
  for (int i=0; i<LEN; i++)
    for (Node * u = bdd[i]; u != NULL; u = u->nxt) {
      node_print(u);
      printf("\n");
```

```
    }
}

void destroy(void) {
  for (int i = 0; i < LEN; i++) {
    Node * u = bdd[i];
    while (u != NULL) {
      Node * v = u->nxt;
      free(u);
      u = v;
    }
  }
}

bool node_eq(Node * u, Node * v) {
  if (u->kind == LEAF)
    return v->kind == LEAF && u->leaf.val == v->leaf.val;
  else
    return v->kind == COMPOUND && u->compound.var == v->compound.var &&
      u->compound.low == v->compound.low &&
      u->compound.high == v->compound.high;
}

int hash_code(Node * u) {
  return u->kind == LEAF ? u->leaf.val :
    (1<<24) + 37*u->compound.var + (1<<10)*23*hash_code(u->compound.low)
    + 19*hash_code(u->compound.high);
}

Node * get_node(Node u) {
  if (u.kind == COMPOUND && u.compound.low == u.compound.high)
    return u.compound.low;
  const int hash = hash_code(&u), pos = (hash < 0 ? -hash : hash)%LEN;
  Node * curr = bdd[pos], * prev = NULL;
  while (curr != NULL) {
    if (node_eq(&u, curr)) return curr;
    prev = curr;
    curr = curr->nxt;
  }
  curr = malloc(sizeof(Node));
  *curr = u;
  if (prev == NULL) bdd[pos] = curr; else prev->nxt = curr;
  curr->nxt = NULL;
  curr->id = size++;
  return curr;
}
```

```
Node * leaf(bool val) {
  return get_node((Node){ .kind = LEAF, .leaf = (struct LeafPart){ val } });
}

Node * compound(int var, Node * low, Node * high) {
  return get_node((Node){ .kind = COMPOUND,
      .compound = (struct CompoundPart){ var, low, high } });
}

Node * not(Node * node) {
  return node->kind == LEAF ? leaf(!node->leaf.val) :
    compound(node->compound.var, not(node->compound.low),
           not(node->compound.high));
}

Node * and(Node * node1, Node * node2) {
  if (node1->kind == LEAF)
    return node1->leaf.val ? node2 : leaf(false);
  if (node2->kind == LEAF)
    return node2->leaf.val ? node1 : leaf(false);
  struct CompoundPart u=node1->compound, v=node2->compound;
  if (u.var < v.var)
    return compound(u.var, and(u.low, node2), and(u.high, node2));
  if (u.var == v.var)
    return compound(u.var, and(u.low, v.low), and(u.high, v.high));
  return compound(v.var, and(node1, v.low), and(node1, v.high));
}

int main() {
  Node *f = leaf(false), *t = leaf(true), *x = compound(1, f, t);
  assert(and(x,not(x)) == f);
}
```