THE UNIVERSITY OF SYDNEY

COMP3109 Assignment 1

Functional Programming (10 marks)

This first assignment is an **individual assignment**, and is due in Week 5 **on Friday, August 29, at 5pm**. Your assignment will not be assessed unless all two of the following criteria are met:

1. Hand in a signed academic honesty form in the tutorial of Week 5.
2. Pack the source code into a zip file and submit it via eLearning. The documentation of the program should be done in form of comments. Please be verbose with your comments, i.e., explain your ideas thoroughly, saying in plain English how you have implemented the program. Not complying to documentation standards will give you less marks for your assignment.

In this assignment you will implement a function evaluator for boolean functions. Please choose one of the following functional programming languages to implement your assignment in: LISP, SCHEME or Haskell. Your assignment should run on the `ucpu[01]` machines.

We define a set of Boolean functions for a variable environment. A variable environment is a list of pairs, each associating a Boolean value with a variable, i.e., an environment is given as:

```
((var1, value1) ... (varN, valueN))
```

We assume that data values in an association list are Booleans, i.e., either the value `#t` or `#f`. Another assumption is that we have for a single variable a unique pair in the variable environment. For example the following variable environment describes the values of variables `a`, `b`, and `c`.

variable	value
a	#t
b	#f
c	#t

This table is translated to following list object:

```
( define my-environment '((a #t) (b #f) (c #t)) )
```

For a variable environment, we would like to formulate a "composed" boolean function. This boolean function queries the environment for variable values, and computes Boolean operations. A Boolean function is given in form of a recursively defined list. The recursive list is an element of the following inductively defined set Q:

1. If `v` is a variable, then `(s-var v)` $\in Q$.
2. If `a`, `b` $\in Q$ then `(s-nand a b)` $\in Q$.
3. If `a` $\in Q$ then `(s-not a)` $\in Q$.
4. Nothing else is in Q.

where the Boolean operator `s-nand` represents the negated-and i.e., $\neg(a \wedge b)$, the Boolean operator `s-not` represents the negation, i.e., $\neg a$, and `s-var` retrieves the value of the given variable from the variable environment.

For example the following list `(s-not (s-nand (s-var a) (s-var b)))` is in Q and semantically models the query: $\neg(\neg(a \wedge b))$.

Task 1 (3 marks)

Write a function `(find-vars <query>)` that recursively traverses a query and returns a list of variables that are used in the query. For example

```
(find-vars '(s-nand (s-nand (s-var a) (s-var b)) (s-var a)))
```

should return `(a b)`. Do not return multiple occurrences of variables. You may assume that the recursive list `<query>` is always an element of Q, so error-handling is not needed.

Task 2 (3 marks)

Write a function `(transformer <query>)` that takes a query in Q as an argument and generates a *function* as output. The generated function has a variable environment as input and returns `#t` (for true) if the predicate of the query holds, otherwise it returns `#f` (for false). Note that the transformer reads recursively the elements of a query and constructs a lambda function on the fly. For example

```
> (define tq (transformer '(s-not (s-nand (s-var a) (s-var b)))))
...
> (apply tq '((a #t) (b #f)))
#f
> (apply tq '((a #t) (b #t)))
#t
```

Hint: Start simple. Try to translate the variable environment access and from there you construct lambda terms for operators "not" and "nand".

Task 3 (4 marks)

This task is to write a query simplifier `(simplify <query>)`, where `<query>` is an element in Q, the set of recursively defined lists. The simplifier produces a new query as a result that is simpler. Find an extensive rule set to simplify queries. Read up about De-Morgan's law. You can start with simple rules such as `(simplify '(s-not (s-not (s-var a))))` that may return `(s-var a)`. Aim for minimizing the number of operators in the result query.