CSE 3302/5307 Lab Assignment 3

Due August 3, 2015

Goals:

1. Understanding of Scheme, especially recursion.
2. Understanding of short-circuit Boolean expression processing by branching.

Requirements:

1. Generalize the function `compile` in the provided Scheme code, `http://ranger.uta.edu/~weems/NOTES3302/LAB/15SUM/LAB3/lab3.rkt`, so that `and` and `or` are no longer restricted to be binary. So, one or more sub-expressions is fine. Only `compile` will require modification. `sc`, `operandCount`, `listing`, and `interpret` will not.

2. Include a ternary `if` operator taking three argument sub-expressions. If the first argument evaluates as true, the result of evaluating the second argument sub-expression is the result of the `if`. Otherwise, the the result of evaluating the third argument sub-expression is the result of the `if`.

   You will need a new simulated two-way branch instruction, `br2`, for implementing `if`. This instruction will have destinations for both true and false situations. `compile`, `listing`, and `interpret` will need modification. `sc` and `operandCount` will not.

3. Submit your Racket source file on Blackboard by 12:45 p.m. on August 3.

Getting Started:

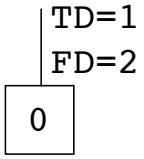
1. `sc` is the simple test driver. It has two arguments: an expression to be compiled to branch instructions and a lat of truth values (`T` and `F`, along with `D` for die) to be consumed by `interpret` when it processes the branch instructions. These truth values are simply matched left-to-right with the `x`'s appearing in the expression.

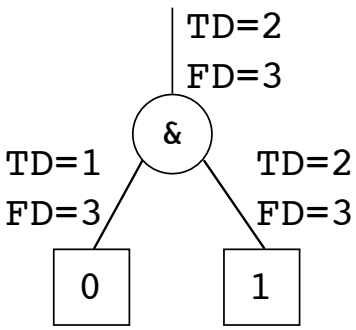
2. `compile` conceptually traverses a tree right-to-left. The examples below are for the provided code. Note that the true destination (for an entire expression) is always the number of `x`'s and the false destination is always one greater. In addition, the number of instructions needed is always the number of `x`'s and the only instructions needed are `brT` (branch on True) and `brF` (branch on False).

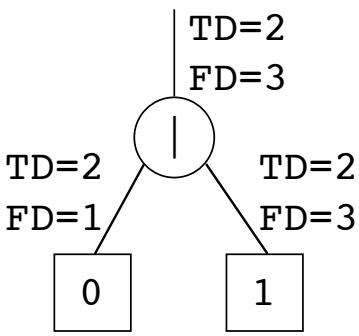
```
> (sc 'x '(T))
x
((0 brF 2))
(T)
At state 0
'("terminate at" 1)
```

```
> (sc '(and x x) '(T F))
(and x x)
((0 brF 3) (1 brF 3))
(T F)
At state 0
At state 1
'("terminate at" 3)
```

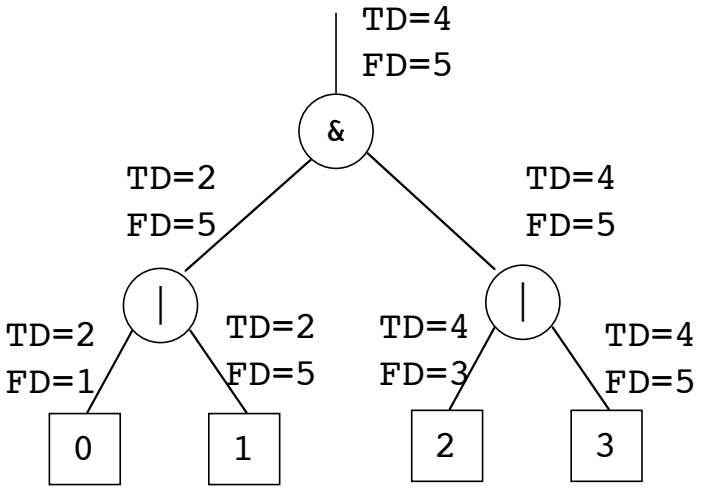
```
> (sc '(or x x) '(F T))
(or x x)
((0 brT 2) (1 brF 3))
(F T)
At state 0
At state 1
'("terminate at" 2)
```



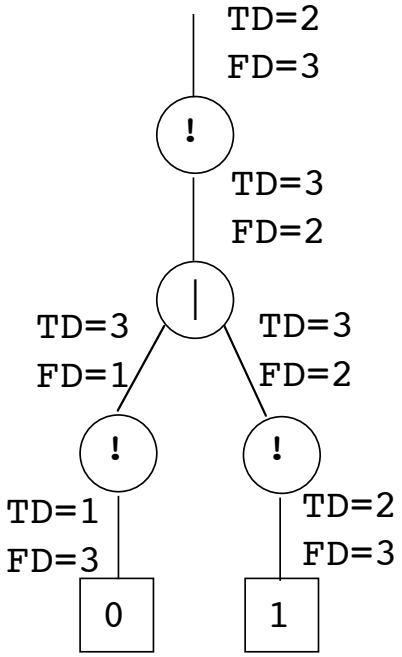




```
> (sc '(and (or x x) (or x x)) '(T F T F))
(and (or x x) (or x x))
((0 brT 2) (1 brF 5) (2 brT 4) (3 brF 5))
(T F T F)
At state 0
At state 2
'("terminate at" 4)
```

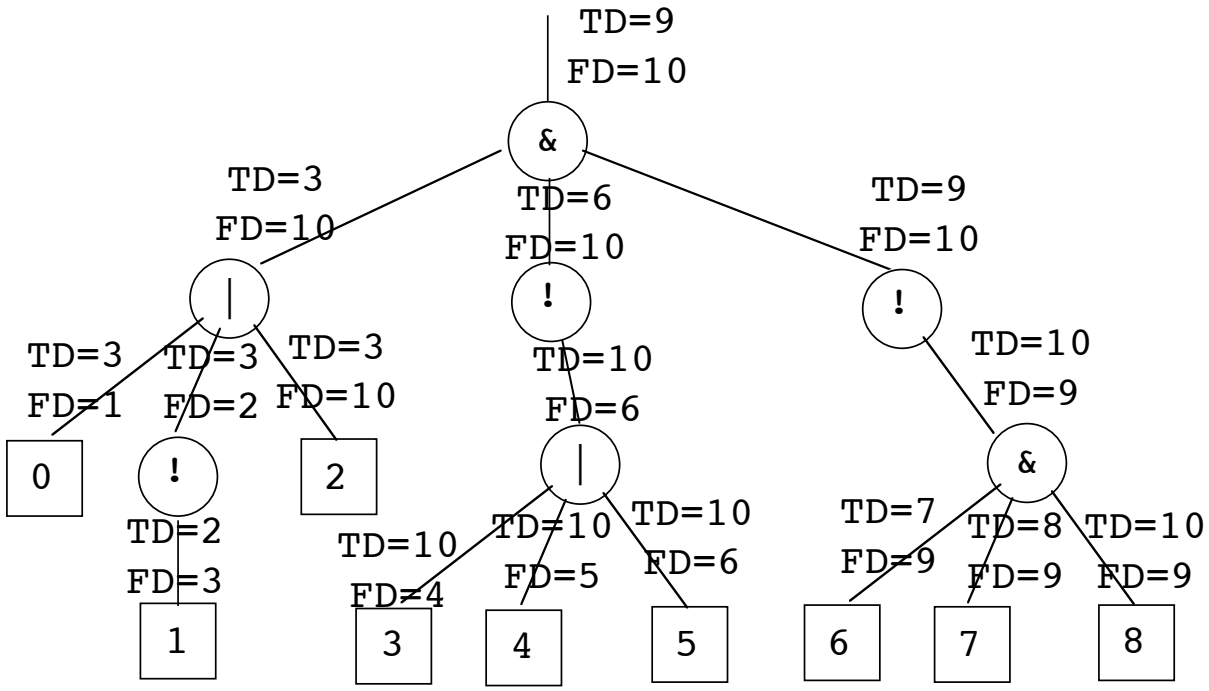



```
> (sc '(not (or (not x) (not x))) '(T T))
(not (or (not x) (not x)))
((0 brF 3) (1 brF 3))
(T T)
At state 0
At state 1
'("terminate at" 2)
```



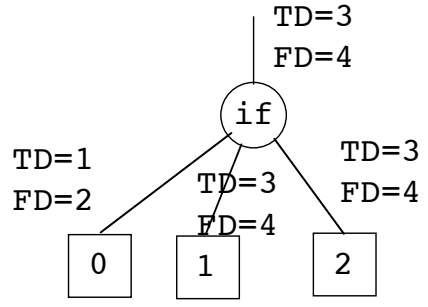
3. This example reflects Requirement 1.

```
(sc '(and (or x (not x) x) (not (or x x x)) (not (and x x x))) '(T T T F F F T T F))
(and (or x (not x) x) (not (or x x x)) (not (and x x x)))
((0 brT 3) (1 brF 3) (2 brF 10) (3 brT 10) (4 brT 10) (5 brT 10) (6 brF 9) (7 brF 9)
   (8 brT 10))
(T T T F F F T T F)
At state 0
At state 3
At state 4
At state 5
At state 6
At state 7
At state 8
'("terminate at" 9)
```

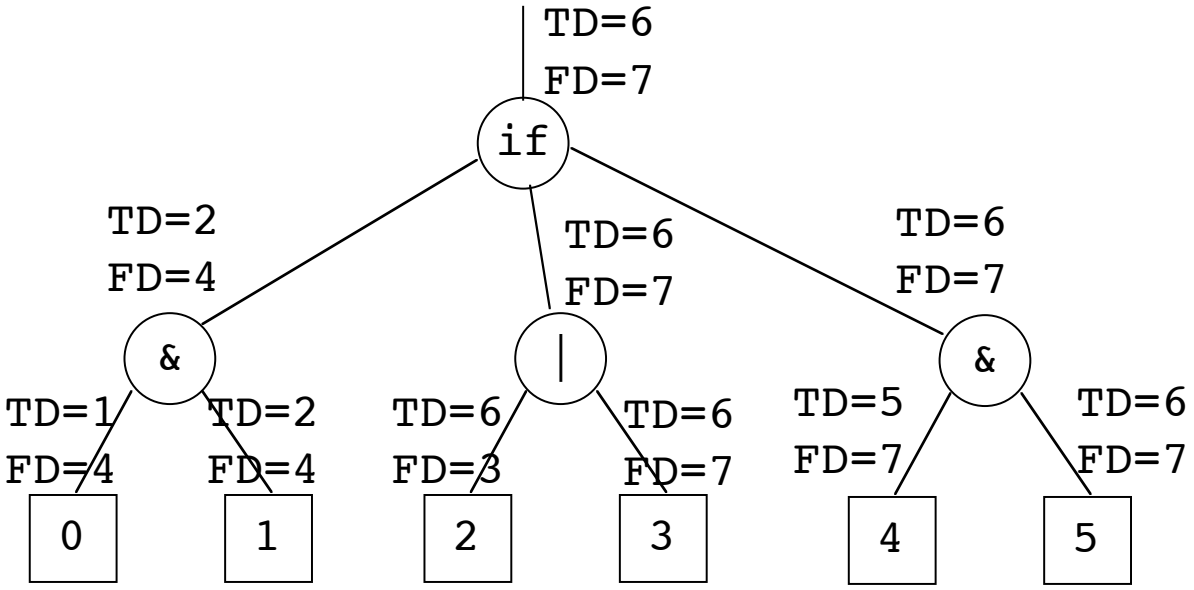


4. These examples reflect Requirement 2.

```
> (sc '(if x x x) '(T F T))
(if x x x)
((0 brF 2 -1) (1 br2 3 4) (2 brF 4 -1))
(T F T)
At state 0
At state 1
'("terminate at" 4)
```



```
> (sc '(if (and x x) (or x x) (and x x)) '(T T T T T T))
(if (and x x) (or x x) (and x x))
((0 brF 4 -1) (1 brF 4 -1) (2 brT 6 -1) (3 br2 6 7) (4 brF 7 -1) (5 brF 7 -1))
(T T T T T T)
At state 0
At state 1
At state 2
'("terminate at" 6)
```



5. The provide code includes a driver, `scLuser`, which does a bit more error checking. You are not expected to generalize it.