CPSC 121 Quiz 2
Wednesday, 2012 July 11

[1] 1. Consider the following (approximate) quote from the Racket documentation: "A branch of a `cond` that starts with `else` must be the last branch."

Translate this to propositional logic using the following definitions: Let e mean "The branch of the `cond` starts with `else`." Let l mean "The branch of the `cond` is the last branch.".

Solution : $e \rightarrow l$

[1] 2. Assuming that 11001 is a 5-bit signed binary number, convert it to decimal:

Solution : It starts with 1; so, it's negative. We flip the bits and add 1 to get $\overline{11001} + 1 = 00110 + 1 = 00111 = 7_{10}$.

Thus, the answer is -7.

[1] 3. Assuming that 00101 is a 5-bit signed binary number, convert it to decimal.

Solution : It starts with 0; so, it's positive. We convert to decimal to get 5.

[2] 4. Convert the binary number $01101100010111110000000000000010$ to hexadecimal.

Solution : First, let's separate it into groups of four: $0110\ 1100\ 0101\ 1111\ 0000\ 0000\ 0000\ 0010$.

Now, we just convert each group of four: 6C5F0002 .

[6] 5. Show—by modeling the programs in propositional logic and using a logical equivalence proof—that the program:

```
;; Boolean Boolean Boolean -> Boolean
(define (is-thingy? b c p)
  (if p
    (not b)
    (if b
      c
      true)))
```

is equivalent to the program:

```
;; Boolean Boolean Boolean -> Boolean
(define (is-thingy? a b c)
  (or (not b) (and (not p) c)))
```

Solution : We model the first piece of code as

$(p \rightarrow \sim b) \wedge (\sim p \rightarrow ((b \rightarrow c) \wedge (\sim b \rightarrow T)))$

We model the second piece of code as

$\sim b \vee (\sim p \wedge c)$

We then prove these equivalent:

Theorem:

$(p \rightarrow \sim b) \wedge (\sim p \rightarrow ((b \rightarrow c) \wedge (\sim b \rightarrow T))) \equiv \sim b \vee (\sim p \wedge c)$

Proof:

$$
\begin{aligned}
(p \rightarrow \sim b) \wedge (\sim p \rightarrow ((b \rightarrow c) \wedge (\sim b \rightarrow T))) &\equiv (p \rightarrow \sim b) \wedge (\sim p \rightarrow ((b \rightarrow c) \wedge (b \vee T))) && \text{by IMP} \\
&\equiv (p \rightarrow \sim b) \wedge (\sim p \rightarrow ((b \rightarrow c) \wedge T)) && \text{by UB} \\
&\equiv (p \rightarrow \sim b) \wedge (\sim p \rightarrow (b \rightarrow c)) && \text{by I} \\
&\equiv (p \rightarrow \sim b) \wedge (p \vee (b \rightarrow c)) && \text{by IMP} \\
&\equiv (p \rightarrow \sim b) \wedge (p \vee \sim b \vee c) && \text{by IMP} \\
&\equiv (\sim p \vee \sim b) \wedge (p \vee \sim b \vee c)) && \text{by IMP} \\
&\equiv \sim b \vee (\sim p \wedge (p \vee c)) && \text{by DIST} \\
&\equiv \sim b \vee (\sim p \wedge p) \vee (\sim p \wedge c) && \text{by DIST} \\
&\equiv \sim b \vee F \vee (\sim p \wedge c) && \text{by NEG} \\
&\equiv \sim b \vee (\sim p \wedge c) && \text{by I}
\end{aligned}
$$

QED

[9] 6. You are working on a new technology for computers in which the basic unit of information is a "trit"—a "trinary digit"—with three values which we label "0", "1", and "2". The basic unit of memory is the "tryte" composed of 9 trits. (You may leave numeric answers as formulas using addition, subtraction, multiplication, division, and exponentiation.)

[1] (a) If we use a base 3 representation for unsigned trinary numbers (like our base 2 representation for unsigned binary numbers), what decimal value would the unsigned trinary number 000000121 represent?

Solution : $1 * 3^2 + 2 * 3^1 + 1 * 3^0 = 1 * 9 + 2 * 3 + 1 = 9 + 6 + 1 = 16$

[2] (b) What's the decimal value of the largest unsigned trinary number representable with one "tryte"?

Solution : Since we do represent 0 before representing 1 and successive positive integers, it will be $3^9 - 1$.

[2] (c) Consider the following proposed representation for *signed* numbers using a tryte: The final 8 trits are an unsigned trinary number. If the first trit is 0, the number is positive (technically, non-negative). If the first trit is 1, the number is negative (technically, non-positive). We disallow the value of 2 for the first trit.

How many "wasted" trit patterns does this representation have? (A pattern is "wasted" if it's disallowed or represents a value that is already representable in another way.)

Solution : Since 2 is disallowed for the first trit, we "waste" all 3^8 patterns of the remaining trits that combine with it. However, we *also* "waste" one trit pattern by representing 0 twice (as -0 and $+0$). That's $3^8 + 1$ "wasted" patterns.

[2] (d) One can easily represent the number $\frac{1}{3}$ using the trinary 0.1; so, it seems as if trinary may solve the problem we had representing fractions with binary numbers.

Give a number that illustrates that trinary does *not* solve this problem (or even make significant progress on it). (You *may* but do *not* need to explain your answer.)

Solution : $\frac{1}{2}$ (and many other numbers) is not representable with a finite number of trits using this scheme. If we subtract 0.1_3 from $\frac{1}{2}$, we're left with $\frac{1}{6}$, but because $\frac{1}{6} * 3 = \frac{1}{2}$, the representation for $\frac{1}{6}$ is the same as that of $\frac{1}{2}$ but with the "trecimal point" shifted left by one place. In other words, $\frac{1}{2} = 0.111\ldots_3$.

[2] (e) Why would hexadecimal be a *poor* choice for a compact representation of trinary numbers?

Solution : One hexadecimal digit has 16 possible values. That corresponds to somewhere *between* 9 and 27 (2 and 3 trits), meaning conversion from "trecimal" to hexadecimal would be complex, not possible by just "splitting into groups" and converting each group.

BONUS: Earn up to 2 bonus points by doing one or more of these problems.

- We have been using propositional logic to model and analyze Racket programs. As always, however, our model is *not* the same as the system it models.

  Give an example of two programs that are likely to behave differently even though they are logically equivalent. Include the sample code, the models of the code, the proof of equivalence, and an explanation of why the code will not behave the same.

  The following entry in the Racket documentation may help.

  > `(time` *expression*`)`
  >
  > Measures the time taken to evaluate *expression*. After evaluating *expression*, `time` prints out the time taken by the evaluation (including real time, time taken by the CPU, and the time spent collecting free memory). The value of `time` is the same as that of *expression*.

Solution : If you know that Racket's `and` is "short-circuiting" (only evaluates the second expression if the first is true), then this is a simple example equivalent up to commutativity:

`(and (time true) false)` and `(and false (time true))`

Notice that in one version we'll see a time printed out but in the other version we won't.

The real point here is that code with "side-effects" is harder to reason about. What's a side-effect? Any effect of an expression other than the value it produces. The most common side-effects you'll run into are input/output (like the printing here) and changes to memory (setting variables' values, which you'll have seen by the end of CPSC 110). Avoiding these—at least when they don't make the code much simpler—is a good idea to make modeling of the code simpler!

(`time` exposes a surprising side-effect. Say it printed nothing but returned the amount of time that *expression* takes to evaluate (not its value). It's still fundamentally about side-effects. For example, if `complex-exp` below has no side effects but takes a long time to compute, are these expressions equivalent: `(time (and complex-exp false))` and `(time false)`? They're *logically* equivalent, anyway.)

Of course, something very similar could be done with `or`.

With a bit more effort, we can turn this into a simple example that uses `if` instead.

- Consider the following representation for signed trinary numbers using a tryte: 0 and 1 will be interpreted as they are for unsigned trinary numbers. A 2 will instead be interpreted as a -1. With this scheme, determine: the largest representable value, the smallest representable value, and the number of "wasted" bit patterns; also, give good algorithms for negating a number and adding two numbers.

Solution : We'll leave the *reasons* for these as an exercise, but: $\frac{3^9-1}{2}$, $-\frac{3^9-1}{2}$, and none; and "change all 1s to -1s and -1s to 1s" and addition we leave as an exercise.

But.. we will note the following: imagine adding two trits and a trit "carry in". The smallest the result can be is -3. The largest it can be is 3. Can we convert that into output of one trit and a trit "carry out"?