
CS 320 Principles of Programming Languages, Spring 2019   Homework 1

Notes and sample solutions

Question 1:

a) "A software developer uses a compiler to translate C code into executable programs that can be run on a Linux computer with an Intel processor.

Syntax: The text files that hold the C source code for the program and the binary files that match the format expected for executable programs on an Intel CPU

Semantics: The meaning of the compiled program should be the same as the meaning of the source program; a correct compiler should map valid inputs to valid outputs, always preserving the semantics.

b) "An innovative Portland start up is using artificial intelligence to generate two sentence summaries of news articles that are published on major web sites and then share the results with the subscribers to its mailing list."

Syntax: The conventions of the natural language (English, Spanish, Japanese, etc.) that the original articles are written in, as well as the language in which the summaries are produced. The HTML syntax that is used to construct the web pages for the input articles may also be relevant here.

Semantics: The output is only a summary of the original article, so it likely cannot contain all the information that was in the original and hence will not have the same semantics. But the summary should be consistent with the original article---the semantics of the former should be a "subset" of the semantics of the latter---and, ideally, the summary will identify the most important details of the original. (A nice goal in theory, but highly subjective, and very difficult to formalize and/or implement!)

c) "The tenant in a recently constructed house is able to use a voice activated assistant (think of something like Siri, Hey Google, Amazon Alexa, or Microsoft Cortana) to turn on the lights in their home without having to touch a switch."

Syntax: The sound that is produced when the tenant speaks, and the formats that are used to represent it as digital data, including raw audio and phonetic or textual versions that are generated by speech recognition software.

Semantics: The behavior of the automated home in response to the user request, such as the action of turning on the lights.

d) "The IRS allows people to submit the information for their tax returns via an online system. An advantage for taxpayers is that the system gives them a prompt notification if it finds any errors in their return, and then provides an opportunity for them to submit a corrected version."

Syntax: The web form that users complete to provide the information to the IRS, and the format that is used to report errors back to users.

Semantics: The interpretation of the tax laws that (in theory) will allow us to compute the amount of tax that is owed or that should be refunded as a function of the data provided by each person using the system.

Question 2:

a) Every truth table for a formula of the specified kind takes the form:

```
   A   |   B   | Out
 ------+-------+-------
 False | False |  ?
 ------+-------+-------
 False | True  |  ?
 ------+-------+-------
 True  | False |  ?
 ------+-------+-------
 True  | True  |  ?
```

where each of the "?" entries in the rightmost column must be filled in with either False or True. Because there are two options for each of four empty cells, there are a total of 2 x 2 x 2 x 2 = 16 possible truth tables.

To simplify the amount of text that I had to type in the following, I added the following definitions to PropScratch.lhs:

```
  a  = VAR "A"
  b  = VAR "B"
  na = NOT a
  nb = NOT b
```

Now I can write formulas like AND a b instead of AND (VAR "A") (VAR "B"), which also makes the text a little easier to read.

Now let's proceed to enumerate all of the possible truth tables, with an associated abstract syntax expression for each one. We could do this by thinking of each of the different options as corresponding to a four bit binary number with one digit for each column. However, I'll approach it here instead by categorizing different truth tables by the number of True and False values in each column.

For starters, there is precisely one truth table with zero True entries (and, in a similar way, only one truth table with four True entries); to satisfy the requirement that our Prop values include both "A" and "B", we use (AND a b) as an input to a logic gate whose output is fixed by the value of the other input:

```
PropScratch> truthTables [ AND FALSE (AND a b), OR TRUE (OR a b) ]
  A   |   B   |       |
------+-------+-------+-----
False | False | False | True
------+-------+-------+-----
False | True  | False | True
------+-------+-------+-----
True  | False | False | True
------+-------+-------+-----
True  | True  | False | True
```

There are four truth tables in which there is only one True entry, all of which can be produced using an AND operation with different combinations of {a,na} and {b,nb}:

```
PropScratch> truthTables [ AND a b, AND a nb, AND na b, AND na nb ]
  A   |   B   |       |       |       |
------+-------+-------+-------+-------+------
False | False | False | False | False | True
------+-------+-------+-------+-------+------
False | True  | False | False | True  | False
------+-------+-------+-------+-------+------
True  | False | False | True  | False | False
```

```
------+-------+-------+-------+-------+------
True  | True  | True  | False | False | False
```

If we replace the ANDs in the last example with ORs, then we get truth tables with precisely three True values in each column; again, there are exactly four of these:

```
PropScratch> truthTables [ OR a b, OR a nb, OR na b, OR na nb ]
  A   |   B   |       |       |       |
------+-------+-------+-------+-------+------
False | False | False | True  | True  | True
------+-------+-------+-------+-------+------
False | True  | True  | False | True  | True
------+-------+-------+-------+-------+------
True  | False | True  | True  | False | True
------+-------+-------+-------+-------+------
True  | True  | True  | True  | True  | False
```

This just leaves us to find truth tables with precisely two False and two True values in the output column. Having listed ten of the sixteen possible truth tables above, we know that there must be six such tables. (To double check, we can calculate the same value by taking the number of combinations of two things (the slots where True values will be placed) chosen from a total of four (the slots in the output column of the truth table).)

The next table shows three such formulas, one corresponding to XOR, one in which the outputs are just a copy of A, and one in which the outputs are just a copy of B:

```
PropScratch> truthTables [ xor a b, AND a (OR a b), AND b (OR a b) ]
  A   |   B   |       |       |
------+-------+-------+-------+------
False | False | False | False | False
------+-------+-------+-------+------
False | True  | True  | False | True
------+-------+-------+-------+------
True  | False | True  | True  | False
------+-------+-------+-------+------
True  | True  | False | True  | True
```

The xor a b example here uses a function defined in PropScratch, but it would also be possible to use the equivalent formula OR (AND a nb) (AND na b) directly. Written out in full, this expands to:

```
OR (AND (VAR "A") (NOT (VAR "B")))
   (AND (NOT (VAR "A")) (VAR "B"))
```

The remaining three truth tables can be produced by using the negated versions of these same three formulas: (for readability, I split the list of formulas across three lines)

```
PropScratch> truthTables [ NOT (xor a b),
                           NOT (AND a (OR a b)),
                           NOT (AND b (OR a b)) ]
  A   |   B   |       |       |
------+-------+-------+-------+------
False | False | True  | True  | True
------+-------+-------+-------+------
False | True  | False | True  | False
------+-------+-------+-------+------
True  | False | False | False | True
------+-------+-------+-------+------
True  | True  | True  | False | False
```

This completes the task of enumerating all possible truth tables of the form described in the question.

---

b) There are infinitely many different formulas for any given truth table. To see why this is the case, suppose that p is a formula corresponding to a particular truth table, and then note that each of the following formulas will produce exactly the same truth table:

```
p                                                -- original formula
NOT (NOT p)                                      -- 2 extra NOTs
NOT (NOT (NOT (NOT p)))                          -- 4 extra NOTs
NOT (NOT (NOT (NOT (NOT (NOT p)))))              -- 6 extra NOTs
NOT (NOT (NOT (NOT (NOT (NOT (NOT (NOT p))))))) -- 8 extra NOTs
...                                              -- and so on ...
```

There are infinitely many formulas in this list, each of them different from all of the others (because each one of them has a different number of NOT nodes in its AST)

The discrepancy between Parts (a) and (b) --- infinitely many formulas, but only finitely many truth tables --- occurs because there are many equivalences between syntactically distinct terms that have the same semantics. Each of the examples above relies on the fact that any expression of the form NOT (NOT p) must be equivalent to the smaller expression p. But there are many other equivalences between propositional formulas/circuits that we could have used instead to illustrate the same point. For example, all of the following examples have the same truth table as (AND a b):

```
   AND b a
   AND (AND a a) b
   NOT (OR (NOT a) (NOT b))
   AND (OR FALSE a) (AND b TRUE)
   ...
```

Question 3:

There are essentially four different ways in which we can reduce an expression using the rules from the lecture / used in the materials for lab1:

- Replace a variable with a value, taken from an environment.

- Replace an expression AND l r where l and r are either TRUE or FALSE, with a corresponding TRUE or FALSE result.

- Similar to the previous case, but using OR instead of AND.

- Similar to the previous case, but using NOT instead of OR.

Conversely, an expression is in "normal form" if none of the above cases applies. In particular, this can only occur if the expression is TRUE or FALSE or if it contains at least one variable whose value is not defined in the environment.

NOTE: for each of the parts below, we include a section labeled "Exploration using lab materials"; this is not something that we expect to see in student solutions, but may be helpful in showing how the Prop tools introduced in labs could be used to explore the questions raised in each part.

a) For any integer n>0 it IS possible to construct a Prop value t that reduces to normal form in exactly n steps.

   Justification: Informally, we can see that an expression of the form NOT (NOT (... (NOT TRUE) ...)) with exactly n NOT operators will take exactly n steps to reduce: the first step will reduce the inner (NOT TRUE) with FALSE, leaving an expression with (n-1) NOT operators applied to FALSE. After a further (n-1) steps of the same kind, we will reach a normal form that is either TRUE or FALSE, depending on whether n was even or odd, respectively. This argument can be made more precise by reformulating it as a proof by induction, but doing that is not necessary for the purposes of this question.

Note that there are many other ways to constuct Prop values with the properties required here; the method above is just one option, but that is still sufficient to justify the original claim.

Exploration using lab materials: We can use the "normalize" function to calculate the length of a reduction sequence for the first few expressions in the sequence above, observing that the result increases by one for every NOT that we add:

```
PropScratch> length (normalize [] TRUE)
1
PropScratch> length (normalize [] (NOT TRUE))
2
PropScratch> length (normalize [] (NOT (NOT TRUE)))
3
PropScratch> length (normalize [] (NOT (NOT (NOT TRUE))))
4
PropScratch>
```

In fact, we can calculate the lengths of the reduction sequences for the first 16 such formulas using the following Haskell expression:

```
PropScratch> take 16 (map (length . normalize []) (iterate NOT TRUE))
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
PropScratch>
```

This uses features of Haskell that have not been covered in lectures, so it is beyond the scope of this assignment and we do not expect to see this in student solutions. Nevertheless, some students, especially those who have previous experience with Haskell, may be interested to see this example.

b) It is NOT possible to construct a term that will produce an infinite sequence of steps using the reduction process that was described in the lectures.

Justification: Every reduction step e1 ==> e2 produces an output expression e2 that either has fewer variables than e1, or else has fewer tree nodes than e1. (The latter occurs for those reductions the eliminate an AND, OR, or NOT applied to known input values.) Whatever expression we might start with, there can only be a finite number of variables, and a finite number of tree nodes, so reduction can only continue for some finite number of steps before there are no remaining variables with known values and no remaining operators with known arguments. At this point, the process has reached a normal form and must stop.

Exploration using lab materials: As in the previous part, we can use a combination of "length" and "normalize" to calculate the length of the reduction sequence for any Prop value t by evaluating expressions of the form: length (normalize [] t). For example:

```
PropScratch> length (normalize env0 (AND a (OR b a)))
6
PropScratch>
```

(I'm using a and b as shorthands for VAR "A" and VAR "B" respectively, so the Prop values that we're using here do still satisfy the restrictions in the question text.)

No matter how complicated we make the formula, this process always terminates and prints a finite number of steps. This in itself is not a proof of the general result, but it does encourage us to believe that there are no examples that require an infinite number of steps, and to develop a more general argument like the one above.

c) It is NOT possible to construct a term that could produce two distinct normal forms when used as an input to a normalization process.

Justification: If there are two (or more) different ways to reduce a given term, then there must be a node of the form AND p q or OR p q in the tree, where one of the possible reductions applies to p (or a subexpression of p) and one applies to q (or a subexpression of that). In this situation, we can perform the reduction in p and the reduction in q in either order: rewriting the left expression will not modify the right expression, and hence will not prevent us from rewriting the right expression. By reordering the steps in a reduction sequence like this, one pair at a time, we can convert any reduction sequence into any other, without changing the final result. It follows therefore that the final results produced by each reduction sequence must be the same.

Exploration using lab materials: Again, we can use the lab materials to test some examples and develop some intuitions that lead to the conclusions described above. For example, for any expression t and a suitable environment env, we can calculate the number of different normal forms that are possible using: length (normalForms env t), and we can calculate the number of *distinct* normal forms using: length (nub (normalForms env t)). No matter what example t we try, the latter expression always returns 1, even if the former shows that there are many possible reduction sequences leading to a normal form. For example:

```
PropScratch> length (normalForms env0 (AND (OR a b) (OR b a)))
80
PropScratch> length (nub (normalForms env0 (AND (OR a b) (OR b a))))
1
PropScratch>
```
