Computer Science 52

Assignment 6

Due Friday, October 27, 2017, at 5:00 PM

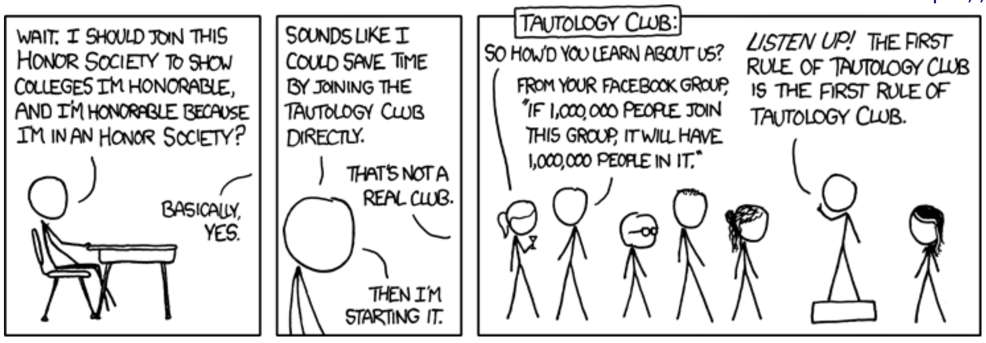
Important note on scheduling Because of the intervening Fall Recess, we have a bit of overlap on Assignments 6 and 7. The former is a relatively short assignment that would normally be due on October 20, but we are giving you a little extra flexibility and making it due a week later. However, Assignment 7 is will be due one week later, on November 3, and it is a a more substantial assignment. We strongly encourage you to finish Assignment 6 early and move on to Assignment 7!

Reading In the last assignment, you built logical circuits using gates. In this assignment, you will carry out similar operations by manipulating logical formulas. Before starting to write code, take some time to study Sections VI and VII of the course document *Bits and Logic* and learn about boolean formulas and their representation in SML.

Preparation and submission As usual, copy the template and check files from the course Calendar page, and rename the template file as `asgt06.sml`. There is also a file `asgt06-examples.sml` which contains sample formulas that you will find helpful when testing your code. When you have completed the assignment and successfully run the check file, upload your file `asgt06.sml` through the course submission page. Submit *only* that one file.

The assignment This assignment continues our exploration of functional programming, recursion and SML. You are now ready to work in a richer context and apply your skills to larger and more complex

https://xkcd.com/703/


WAIT. I SHOULD JOIN THIS HONOR SOCIETY TO SHOW COLLEGES I'M HONORABLE, AND I'M HONORABLE BECAUSE I'M IN AN HONOR SOCIETY?
BASICALLY, YES.
SOUNDS LIKE I COULD SAVE TIME BY JOINING THE TAUTOLOGY CLUB DIRECTLY.
THAT'S NOT A REAL CLUB.
THEN I'M STARTING IT.
TAUTOLOGY CLUB:
SO HOW'D YOU LEARN ABOUT US?
FROM YOUR FACEBOOK GROUP, "IF 1,000,000 PEOPLE JOIN THIS GROUP, IT WILL HAVE 1,000,000 PEOPLE IN IT."
LISTEN UP! THE FIRST RULE OF TAUTOLOGY CLUB IS THE FIRST RULE OF TAUTOLOGY CLUB.

problems. Your task is to convert a boolean formula into an equivalent *disjunctive normal form.*

In propositional logic, a *literal* is a propositional variable or its negation. A *clause* is the conjunction of literals, and a *disjunctive normal form*, abbreviated DNF, is a disjunction of clauses. Every formula of propositional logic is equivalent to one in DNF.

Note the similarity with circuits constructed with the minterm expansion. The input lines or their negations (literals) are fed to and-gates (creating clauses) whose outputs are sent to a single or-gate. The final output is a disjunction of clauses.

Here is a simple example. We will carry along a richer example through the actual problem statements. Using one of the identities in Table 1 of *Bits and Logic,* the formula

$$p_0 \Rightarrow (p_1 \Rightarrow p_0)$$

can be transformed into a n equivalent DNF formula,

$$\neg p_0 \vee \neg p_1 \vee p_0.$$

There are three clauses, each with one literal.

The template file contains the `booleanExpression` datatype which will allow us to represent formulas in SML. In that notation, the original formula is

```
Implies (Prop 0, Implies (Prop 1, Prop 0))
```

and the DNF equivalent is

```
Or (Not (Prop 0), Or (Not (Prop 1), Prop 0))
```

We will use a slightly different representation for the DNF formulas: A clause will be a list of literals, and a DNF formula will be a list of clauses. With that convention, our DNF formula with three clauses is

```
[ [Not (Prop 0)], [Not (Prop 1)], [Prop 0] ]
```

As you work, keep your code clean and simple. Use the propositional identities to maintain logical correctness, and stay true to the SML datatypes. The function `evalExp` that is described in Section VII of *Bits and Logic* and repeated in the template file is a good example.

1. [4 points] Write a function `elimConnectives` that converts a boolean expression into an equivalent one that uses only `Not`, `And`, and `Or`. Follow the recursive pattern of the function `evalExp` in the template file, and use these equivalences to make the transformations.

$$\begin{aligned} A \oplus B &\equiv (A \,\&\, \neg B) \vee (\neg A \,\&\, B) \\ A \Rightarrow B &\equiv \neg A \vee B \\ A \Leftrightarrow B &\equiv (A \,\&\, B) \vee (\neg A \,\&\, \neg B) \end{aligned}$$

Here is an example that we will use throughout the problems. The formula

$$(p_0 \Leftrightarrow p_1) \Rightarrow p_2$$

is transformed into

$$\neg((p_0 \mathbin{\&} p_1) \vee (\neg p_0 \mathbin{\&} \neg p_1)) \vee p_2.$$

Analogously, in SML, applying `elimConnectives` to

```
Implies (Iff (Prop 0,Prop 1),Prop 2)
```

yields

```
Or (Not (Or (And (Prop 0, Prop 1),
             And (Not (Prop 0), Not (Prop 1)))),
    Prop 2)
```

Use `evalExp` as a model, rely on recursion, and do not attempt any simplification at this stage.

```
elimConnectives : booleanExpression -> booleanExpression
```

2. [5 points] The next step is to transform a boolean formula by pushing all the negation operators as deep into the formula as possible. It is also a convenient time to eliminate double negations. The De Morgan laws are the crucial facts we need. Write a function `applyDeMorgan` that uses these identities.

$$\begin{aligned}
\neg \mathit{true} &\equiv \mathit{false}\\
\neg \mathit{false} &\equiv \mathit{true}\\
\neg\neg A &\equiv A\\
\neg(A \mathbin{\&} B) &\equiv \neg A \vee \neg B\\
\neg(A \vee B) &\equiv \neg A \mathbin{\&} \neg B
\end{aligned}$$

Assume that the argument presented to your function has already been processed by `elimConnectives`. Declare an exception `ConversionException` and raise it if a connective other than `Not`, `And`, or `Or` is encountered. Continuing with the example, we see that the formula

$$\neg((p_0 \mathbin{\&} p_1) \vee (\neg p_0 \mathbin{\&} \neg p_1)) \vee p_2$$

is transformed into

$$((\neg p_0 \vee \neg p_1) \mathbin{\&} (p_0 \vee p_1)) \vee p_2.$$

As an SML data structure, the result of applying `applyDeMorgan` is

```
Or (And (Or (Not (Prop 0), Not (Prop 1)),
         Or (Prop 0, Prop 1)),
    Prop 2)
```

```
applyDeMorgan : booleanExpression -> booleanExpression
```

Your function will have more patterns—and ones with more complicated structure—than you saw in the `evalExp` example.

3. [5 points] The final step in the conversion is to produce an equivalent DNF formula. Write a function `toDNF` that takes an expression

that has been processed by `elimConnectives` and `applyDeMorgan`, and produces the DNF formula.

Recall that a *literal* is a proposition letter or the negation of a proposition letter. A *clause* is a conjunction of zero or more literals, and a *DNF formula* is a disjunction of zero or more clauses. Instead of writing DNF formulas as boolean expressions, it is convenient to represent them as lists of lists of literals. That is, a formula is a list of clauses, and each clause is a list of literals.

Now is a good time to think carefully about boundary cases. A clause is a list of literals. What is the truth value of the empty clause? A formula is a list of clauses. What is the truth value of the empty formula?

The formula that we saw as the result of the example in Problem 2

$$((\neg p_0 \vee \neg p_1)\,\&\,(p_0 \vee p_1)) \vee p_2$$

would give the result

$$(\neg p_0 \,\&\, p_0) \vee (\neg p_0 \,\&\, p_1) \vee (\neg p_1 \,\&\, p_0) \vee (\neg p_1 \,\&\, p_1) \vee p_2$$

or, in SML,

```
[ [Not (Prop 0),Prop 0],
  [Not (Prop 0),Prop 1],
  [Not (Prop 1),Prop 0],
  [Not (Prop 1),Prop 1],
  [Prop 2] ].
```

Here is a framework for your function.

```
fun toDNF True                =
  | toDNF False               =
  | toDNF (Prop j)            = [[Prop j]]
  | toDNF (Not (Prop j))      = [[Not (Prop j)]]
  | toDNF (Or (exp0, exp1))   =
  | toDNF (And (exp0, exp1)) =
  | toDNF _                   = raise ConversionException;
```

The code you write will be relatively short—you have only four cases to fill in. But you will have to think deeply about each one. It is important that each case be exactly correct. Spend some time thinking about the desired results as lists of lists of literals. For the `And` connective, you will use the distributive laws.

An illustrative exercise, to see what needs to be done in the `And` case, is to expand $(R \vee S \vee T)\,\&\,(U \vee V)$ into a disjunction of conjunctions.

$$(A \vee B)\,\&\,C \equiv (A \,\&\, C) \vee (B \,\&\, C)$$
$$A\,\&\,(B \vee C) \equiv (A \,\&\, B) \vee (A \,\&\, C)$$

Again, avoid the temptation to simplify the resulting formulas—at least for the moment.

```
toDNF : booleanExpression -> booleanExpression list list
```

4. [2 points] Combine the results of the preceding problems to declare a function `expToDNF` which takes an arbitrary boolean expression and produces an equivalent DNF formula.

```
expToDNF : booleanExpression -> booleanExpression list list
```

Your function will be a simple composition of the three functions you have already written. From end to end, our ongoing example $(p_0 \Leftrightarrow p_1) \Rightarrow p_2$ is transformed into

$$(\neg p_0 \,\&\, p_0) \vee (\neg p_0 \,\&\, p_1) \vee (\neg p_1 \,\&\, p_0) \vee (\neg p_1 \,\&\, p_1) \vee p_2.$$

In the notation of the SML data structure, the result of applying `expToDNF` to

```
Implies (Iff (Prop 0, Prop 1), Prop 2)
```

is a list of lists of literals:

```
[ [Not (Prop 0),Prop 0],
  [Not (Prop 0),Prop 1],
  [Not (Prop 1),Prop 0],
  [Not (Prop 1),Prop 1],
  [Prop 2] ]
```

Simplification Before moving on to the next part of the assignment, take some time to experiment with your function `expToDNF`. Try it on some of the examples given in the file `asgt06-examples.sml`. Convert the formulas to DNF, and in the simpler cases, examine the results to convince yourself that they are correct. For the more complicated cases, there is a function in the template file that will verify that your result is indeed a DNF formula and is equivalent to the original formula. You can write something like this:

```
checkEquivalence someFormula (expToDNF someFormula);
```

The result will be `true` if your `expToDNF` function is working correctly.

Be aware that, for a given boolean formula, there is no single "right answer." Many DNF equivalents are possible. Because we have so far discouraged you from simplifying expressions, you will no doubt see some very large DNF formulas. Here are some of the reasons that the results are so long and complex.

- A literal may appear more than once in a clause. Nothing is gained by repetition.
- A clause may contain both a proposition letter and its negation. Such a clause will never evaluate to *true* and may be eliminated from the DNF formula.
- A clause may be repeated in the DNF formula. Duplicate copies may be eliminated. This situation is a little difficult to detect because the literals may appear in different orders in the clauses.

- There may be two clauses C and $\mathcal{D}$ which have all the same literals except that a proposition letter p_k appears in one and its negation in the other. The two clauses can be discarded and replaced with a single clause containing the shared literals. (Why?)
- There may be two clauses C and $\mathcal{D}$ with the property that every literal in C is also in $\mathcal{D}$. Whenever $\mathcal{D}$ is *true*, the clause C will also be *true*, and so $\mathcal{D}$ may be discarded.

The list is not exhaustive. Simplifying a formula, or even determining if it can be simplified, is a computationally intensive process. In this last part of the assignment, we will be satisfied by making some progress toward simplification.

5. [4 points] Write a revision of your function `expToDNF` which carries out some or the simplifications suggested by the list above. Name your function `betterExpToDNF`. Although it is possible to create the DNF formula and then simplify it—say by reducing some of the redundancies, it is more efficient to avoid introducing them in the first place. It is probably enough to declare a new version of `toDNF` and incorporate it into the framework of Problem 4.

```
betterExpToDNF : booleanExpression -> booleanExpression list list
```

Our example result

```
[ [Not (Prop 0),Prop 0],
  [Not (Prop 0),Prop 1],
  [Not (Prop 1),Prop 0],
  [Not (Prop 1),Prop 1],
  [Prop 2] ]
```

contains two clauses that can never be satisfied, so the result simplifies to

```
[ [Not (Prop 0),Prop 1],
  [Not (Prop 1),Prop 0],
  [Prop 2] ].
```

We can measure the complexity of a DNF formula in (at least) two ways: by counting the number of clauses and by counting the total number of literals in all the clauses. There are no firm benchmarks. We will test your function on a variety of formulas, and you will receive full credit if your function `betterExpToDNF` produces logically equivalent DNF formulas in *all* of the cases, and it significantly reduces the complexity over `expToDNF` in *some* of the cases.

The formulas generated by `circularFormula` and `oddParity` in the file of examples are good ones for experimentation. Some formulas, like those generated by `exponentialFormula`, do not permit a reduction in the number of clauses. Pierce's formula, also in the example file, provides a good short test of your methods.