Proposed final assignments

I will provide you with:

- A basic set of rewrite rules
- An idea on how to generate source and target expressions that should guarantee that at least one of those unconditional rewrite rules can be used
- A set of ‘challenge’ rules.

You will:

- **parse** the set of unconditional rules I gave
- **generate expressions** based on the idea
- create an **‘evaluate’ function** that can calculate values for those expressions to test that they and the rules are valid
- **generate a proof** using the rules and expressions
- **Create a cs30 exercise** in which students need to (one of these, as mentioned in the assignment):
  - Determine the correct order for the proof
  - Determine what justifies a certain step (choose from set of options)
  - Find an error you purposefully put into the proof in one of the steps

There are 81 points in total for this assignment.
50 points (10 each) are given for the bold parts listed above.
Another 11 points is given for your use of git.
The final 20 points is awarded for creativity and doing hard things outside of the ‘standard’ exercise.

The assignments with an * at the end of their title are exceptionally challenging.

Sets: conversion to and from set-builder notation

In this assignment, you build an algorithm that takes a symbolic description of a set and turns it into a single set-builder notation description of that same set. Note that 'set-builder notation' is a ternary expression in which a variable gets bound: { … | … . …}
To simplify things, we will always use 'e' to stand for the bound variable, we will always assume the first part to be {e |, and we combine the binding of 'e' with the properties on e. Hence set-builder expressions are of the form: {e | properties on e}
The properties part is a predicate logic formula, which use a different expression language.

The expression language for sets includes:

\cap:	intersection
\cup:	union
\setminus:	set difference
\P:	powerset
A:	a set variable
\left\{e | p\right\}:	set-builder notation, where p is a property in e, and the character 'e' is fixed.

The expression language for properties:

\wedge:	and
\vee:	or
e \in A:	element-of, where A is some set
e \notin A:	not an element of, where A is some set
e \subsetof A:	a subset of, where A is some set

Rules you should use are:

(for all A, B)	A \cap B	= \left\{e | e \in A \wedge e\in B\right\}
(for all A, B)	A \cup B	= \left\{e | e \in A \vee e\in B\right\}
(for all A, B)	A \setminus B	= \left\{e | e \in A \wedge e\notin B\right\}
(for all A)	\P(A)	= \left\{e | e \subseteq A\right\}
(for all p)	e \in \left\{e | p\right\}	= p

To generate an interesting proof, generate an expression for sets on a number of variables using only the \cap, \cup, and \setminus constructors. After applying the rewriting rules above, you should end up with a term that no longer uses any of those constructors. This proof can be presented to the user in either direction (bottom to top or top to bottom). The user should be asked to put the proof in the right order. I'll make some helper functions available for that at the start of week 8.

There are several additions you might think of that would make this exercise harder to generate:
Modest challenge: generate expressions that contain a powerset symbol such that the expressions make sense (for instance, A and powerset(A) aren't compatible, so A \cap powerset(A) shouldn't occur)
Hard challenge: remove double occurrences of "e\in A", that is: "e \in A \wedge e\in A" and "e \in A \vee e\in A" both rewrite to "e \in A", even in the case that there is a middle expression ("e \in A \wedge e \notin B \wedge e \in A"). This requires you to build in some modest reasoning about \vee and \wedge.
Extreme challenge: deal with multiple variables in the set-builder notation, so that the cartesian product A \times B can be translated as well.

Sets: cardinalities

In this assignment, you build an algorithm that takes a symbolic description of a set and computes its size. This means we deal with numbers, a numerical expression consists of:

- the cardinality symbol | … | where … stands for a set-expression
- addition, subtraction, powers of two, and multiplication
- constants (0, 1, 2, …)

A set-expression consists of:

- a basic set variables: A, B, C … (of which the cardinality is presumed to be known)
- expressions that don't propagate nicely: union, intersection, set-difference
- expressions that do propagate nicely: powerset, product

As an initial term generate a set-expression using basic set variables and operations on it. You then generate a proof using these rules:
|A \cup B| = |A| + |B| - |A \cap B|
|A \times B| = |A| \cdot |B|
|\P(A)| = 2^{|A|}
|A \setminus B| = |A| - |A \cap B|
rules about (natural) numbers with +, -, 0, 1, 2, and ^.

After applying these rules, you should end up with a term that has only intersections of 1 or more sets within each |…| expression. You can slightly rewrite this by applying a canceling operation 'on both sides of the equation':
| set-expr | = term1 + term2 + term3
becomes:
| set-expr | - term3 = term1 + term2
The question you ask the user then is: compute '| set-expr | - term3'. The information you give the user is: '|B| = …', '|C| = …' etc (for B and C occuring in term1 and term2).

Coming up with reasonable values for |A|, |B| and |A \cap B| can be done through a truth-table-like technique (to avoid saying |A| = |B| = 1 and |A \cap B| = 2, which shouldn't occur).

There are several additions you might think of that would make this exercise harder to generate:
Modest challenge: generate expressions that contain a powerset symbol such that the expressions make sense (for instance, A and powerset(A) aren't compatible, so A \cap powerset(A) shouldn't occur)
Hard challenge: remove double occurrences of "e\in A", that is: "e \in A \wedge e\in A" and "e \in A \vee e\in A" both rewrite to "e \in A", even in the case that there is a middle expression ("e \in A \wedge e \notin B \wedge e \in A"). This requires you to build in some modest reasoning about \vee and \wedge.
Extreme challenge: deal with multiple variables in the set-builder notation, so that the cartesian product A \times B can be translated as well.

Logic: rewriting expressions

In this assignment, you are asked to generate a proof that 'pushes down' negations.
Predicate logic expressions consist of:
variables: p, q, r
constants: true, false
and, or, and impliciation: \wedge, \vee, \Rightarrow
negation: \neg

Rules you will use to rewrite these are (\equiv is used instead of = for equivalence here):

$\neg(\neg p) \equiv p$	$p \wedge true \equiv p$	$p \vee true \equiv true$	$p \vee false \equiv p$	$p \wedge false \equiv false$
$p \wedge p \equiv p$	$p \vee p \equiv p$	$(p \Rightarrow q) \equiv \neg p \vee q$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
$p \wedge \neg p \equiv false$	$p \vee \neg p \equiv true$			

The distributivity rules should be used some of the time:

option 1:	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$(q \vee r) \wedge p \equiv (q \wedge p) \vee (r \wedge p)$
option 2:	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	$(q \wedge r) \vee p \equiv (q \vee p) \wedge (r \vee p)$
option 3:	no distributivity rules	

Distributivity allows you to eliminate parenthesis except at the lowest level, at the expense of getting potentially really large expressions.

To generate an interesting proof, generate an expression using variables, negated variables, constants, and, or and implication, and take the negation of that expression.

All of the rules listed above have certain names. The exercise should be for the student to identify which step was used.

There are several additions you might think of:
Easy challenge: don't apply $(p \Rightarrow q)$ unless there is a negation symbol immediately above it.

Modest challenge: generate expressions that guarantee the use of certain proof rules, such that the user gets tested on every proof rule.
Hard challenge: allow the rules of associativity and/or commutativity to be applied exactly when needed to apply one of the rules involving true or false.
Hard challenge: common mistakes students make can be expressed as 'rules' that aren't true, such as: $\neg(p \Rightarrow q) \equiv \neg p \Rightarrow \neg q$, or $p \vee (q \wedge r) \equiv (p \vee q) \wedge r$. Create a second version of your exercise in which students are asked to find the error(s) in a proof (for this second version, the motivation behind each proof step should not be displayed). Ensure (using evaluation) that the step is actually wrong. For example, $p \vee (r \wedge r) \equiv (p \vee r) \wedge r$ is an application of the invalid $p \vee (q \wedge r) \equiv (p \vee q) \wedge r$, but both sides happen to equal $p \vee r$ here, so it's not a proper invalid step.

Logic: inequality problems*

A common CS30 exercise is to prove, using induction, properties like 'n! > n' and 'n! > 2^n' (for n large enough), and '(n+m)! ≥ n!m!'. The resulting proofs by induction are straightforward enough that they can be found automatically if the property on which to do induction is known. However, reasoning with inequalities is quite tricky. For this reason, this exercise will only involve reasoning with inequalities, assuming all numbers are real. Later exercises adaptations will add the use of induction to these exercises.

Normally, we are able to replace equals with equals: if x = y, then f(x) = f(y).
In the case of inequalities, we don't have that x > y implies f(x) > f(y).
In fact, a function f that satisfies this property is called *monotonic*. It is possible to find out if — or under what conditions — a function is monotonic.

For instance:
If y > z, then x \cdot y > x \cdot z; provided x > 0
If y > z, then y \cdot x > z \cdot x; provided x > 0
If y > z, then x \cdot y ≥ x \cdot z; provided x ≥ 0
If y > z, then y \cdot x ≥ z \cdot x; provided x ≥ 0

This means that if we wish to rewrite 'x \cdot y' by 'y > z', we can split our proof into three cases, depending on x. If we can quickly prove that 'x > 0', we omit that part of the proof! If it takes a bit longer, we add a sub-proof. If we cannot prove that 'x > 0', we don't apply the rewrite step.

Your basic rules include the four listed above (about multiplication), and:
If x > y, then x + z > y + z
If x < y, then z / x > z / y; provided z > 0 and x > 0
If x < y, then z / x ≥ z / y; provided z ≥ 0 and x > 0
0 > 1

a - a = 0	a / a = 1	0 * a = 0	0 / a = 0
a * (b * c) = (a * b) * c	a + (b + c) = (a + b) + c		
a - (b - c) = a - b + c	a - (b + c) = a - b - c		
0 + a = a	a + 0 = a	1 * a = a	a * 1 = a

Given an expression that includes a '>' or a '≥' and a list of properties to use (each of the form >, '≥' or = with an expression on either side), present a proof, and ask the user to put the proof in the right order.

Reasoning with monotonicity is already quite complicated, but of course there are multiple extra challenges you might think of.

If you are looking to add something easy, consider dealing with constants:
a * b + c * b = d * b; given that d = a+c could be fully calculated (since a and c are constants)

If you are looking to add something hard, consider dealing with rules that do not simplify anything but still might need to be applied:
n! = n * (n-1)! ; provided n > 0

Probability: generating exercises via proofs

This exercise involves some rewrite rules that tend to make expressions larger. You will apply a certain number of those ‘expanding rewrite rules’ (at ‘random’) to a basic expression (say Pr[A]). Then stop the proof search, giving you a more complicated expression. You turn this into an exercise by taking the more complicated expression that arises at the end of the proof, say “Pr[A \wedge B] + Pr[A | \neg B] \cdot (1 - Pr[B])” and then stating for every term in the expression what the value is, say “Pr[A \wedge B] = 0.15”, “Pr[A | \neg B] = 0.75”, “Pr[B] = 0.6”. You can then ask the student to compute the original expression.

The expressions you have to deal are fractions that may contain:
- the constants 0 and 1
- a probability expression like Pr[event-expression | event-expression]. If this expression is of the form Pr[A | Omega], we just write Pr[A]
- multiplication \cdot, addition +, subtraction -, and division \frac{}{}

Event expressions contain:
- a base event: A, B, C
- ‘and’ \wedge and ‘or’ \vee (intersection and union between event sets)
- event negation \neg (this stands for set difference wrt the set of all events)

Your initial expression is just a probability expression: start with a set of base events and generate two event expressions based on those, and you’ll have yourself a probability expression.
The following rules are the expanding rules:
Definition of conditional probability: Pr[A | B] = Pr[A \wedge B]\cdot Pr[B]
Law of total probability: Pr[A] = Pr[A | B]\cdot Pr[B] + Pr[A | \neg B]\cdot Pr[\neg B]
Bayes’ rule: Pr[A | B] = \frac{Pr[B | A]Pr[A]}{Pr[B]}
Inclusion exclusion: Pr[A \vee B] = Pr[A] + Pr[B] - Pr[A \wedge B]

The following rules simplify forms and should be applied whenever possible (these may be applied deterministically):
DeMorgan: \neg (A \wedge B) = \neg A \vee \neg B
DeMorgan: \neg (A \vee B) = \neg A \wedge \neg B
Double negation: \neg (\neg A) = A
Omega elimination: \Omega \wedge A = A \Omega \vee A = \Omega
Empty-set elimination: \emptyset \vee A = A \emptyset \wedge A = \emptyset
Negation in probability: Pr[\neg A] = 1 - Pr[A]
Difference: Pr[A \wedge \neg B] = Pr[A] - Pr[A \wedge B]

The user can be presented with an exercise where they just need to find the right answer, and the fact that such an answer is computable is guaranteed through the generation of the proof. If the user gave the wrong answer, the full proof can be displayed. A good expression evaluation function is essential here in order to be able to present the user with a consistent / ‘sane’ question (ensuring the final answer will be between 0 and 1, among other things).

As an extra challenge, consider adding these rules as well:
Pr[A \wedge B] = Pr[A] \cdot Pr[B] provided that A and B are mutually independent
Pr[A \vee B] = Pr[A] + Pr[B] provided that A and B are mutually exclusive (equivalently: Pr[A \wedge B] = 0). When used to generate a proof, these extra conditions are mentioned as part of the exercise.

Probability: Expected values

This exercise is about computing the expected value of an expression.

Here are the rules that allow you to compute an expected value:
E[X + Y] = E[X] + E[Y]
E[X - Y] = E[X] - E[Y]
E[X * Y] = E[X] * E[Y] + cov(X,Y)
E[c] = c (if c is a constant)
E[cX] = cE[X] (if c is a constant)
standard rules about addition, multiplication and subtraction

You start by generating an expression from random variables, constants, +, - and *, and then wrap the result in E[…]. After applying the rules above, you should end up with something of a form where any occurrence of E[…] only has a single random variable on the … (and possibly occurences of 'cov').
Those sub-expressions in final form, that is: E[X] where X is a random variable, should become given values in the exercise. This is where your evaluator ties in to this exercise. Modeling random variables for your evaluator is a tricky part of this assignment, so talk to Sebastiaan about it early in the process to make sure you understand how to do a good job on it.

After obtaining a proof of this form:
E[original_expr] = term1 + term2 + term3 + …
use 'subtraction from both sides' to move a random selection of (zero or more) terms from right to left to create a slightly more interesting looking question:
compute: E[original_expr] - term1
answer: term2 + term3 + …

As additional moderate challenge:
Describe 'cov(X,Y)=0' as 'X and Y are independent' whenever a 'cov' term arises, and generate models such that this holds. Even better: get rid of ever displaying 'cov' alltogether by giving the expression of 'E[X * Y]' when needed

As a hard challenge:
Certain operations can be translated by case distinction:
E[|X|] = E[X | X > 0]Pr[X > 0] + E[-X | X<0]Pr[X < 0]
This requires a nontrivial syntax overhaul, but really expands what you can and cannot deal with.

Meta: Expressions based on proofs*

In this assignment, we just use some standard rules about addition, subtraction, multiplication and division, generating standard proofs.
The difficulty in this assignment lies in that we wish to generate expressions that lend themselves well to those rules, generating the initial and final expression as we choose what proof rules to apply.

Let's take these rules as an example:
x + y = y + x
(x + y) + z = x + (y + z)
x + -x = 0

Now suppose we wish to apply the rules listed above in order, then we can come up with a certain number of starting expressions.
For the first rule, our staring expression would be a + b and our proof is: a + b = b + a
Applying the second rule, I have multiple possibilities:
First, I could simply put the 'a+b' part of the proof in 'x', 'y' or 'z'. Choosing y would give: (x + (a+b)) + z = (x + (b + a)) + z = x + ((b + a) + z)
As another extreme, I could substitute either a or b by (x+y)+z. Choosing 'a' would give: ((x+y)+z)+b = b + ((x + y)+z) = b + (x + (y + z))
However, more interesting is to substitute b by (c+d): a + (c + d) = (c + d) + a = c + (d + a)
To apply the third rule, the most interesting thing to do is to let a=-d.
That would give the proof: -d + (c + d) = (c + d) + -d = c + (d + -d) = c + 0

Note that throughout our proof, we have substituted variables to allow certain proof-steps to happen. This gives us a generic way to come up with initial terms and interesting proofs. This can work by using a well-known algorithm called unification, which we can use instead of 'matching' (matching is explained in the book and lectures). Having this opens the door to very rapidly coming up with many kinds of CS30 exercises. In the nearby future, it might also help me to automatically generate proofs by induction (this would be the first step to get that to work).

In this exercise-generator, parameters include the list of rules, and the number of desired proof-steps. The generated exercise can be of the "put these proof-steps in the right order" or "identify each proof-step" forms.

As a modest improvement:
- The less interesting cases, where a+b is substituted for y to give (x + (a+b)) + z, are characterised by the fact that the two proof-steps can be re-ordered wrt each other. If there is a substitution that is not as trivial, choose it and don't consider the other options, to make more interesting proofs more likely.

As a hard improvement:
- Sometimes, after a substitution, a proof-step can be taken earlier, rendering the earlier proof unnatural looking: if we are to add the step 'x + 0 = x' in a non-trivial way to the proof (from before):
  'a + (c + d) = (c + d) + a = c + (d + a)', then we would need to choose a=0, giving:
  '0 + (c + d) = (c + d) + 0 = c + (d + 0) = c+d'. Not wrong, but a more sensical proof would be to bypass the associativity step: '0 + (c + d) = (c + d) + 0 = c+d'. Detect when this happens so you can bypass superfluous steps.

Numbers: modulo n with non-monotonic operations

In CS30, students are asked to become proficient in calculations with modulo. This drill will teach them to reason modulo p symbolically.

Equalities that are used *implicitly* include:

a = b (mod p) *implies* a + c = b + c (mod p)

a = b (mod p) *implies* c + a = c + b (mod p)

a = b (mod p) *implies* a * c = b * c (mod p)

a = b (mod p) *implies* c * a = c * b (mod p)

a = b (mod p) *implies* a ^ c = b ^ c (mod p)

x^{p-1} = 1 (mod p)

Note that normal arithmetic rules (on +,-,*,/,^) still hold. For instance: a + 0 = a

This implies:

a + c*p = a (mod p)

You will build a proof-generator that starts with a randomly generated expression and applies rules like ‘a + c = b + c’. The challenge is to come up with a value for ‘b’ while keeping track of the condition that ‘a = b (mod p)’. This becomes particularly interesting because one will have to nest the proof-generator conditions:

a = b (mod p) *implies* a + c = b + c (mod p)

so (choosing a = x*y and b=z*y):

x*y = z*y (mod p) *implies* (x*y) + c = (x*y) + c (mod p)

but we know that x*y = z*y (mod p) if x = z (mod p), so:

x = z (mod p) *implies* (x*y) + c = (z*y) + c (mod p)

Hence upon finding the term ‘(x * y) + c’, we might rewrite this to ‘(z * y) + c’ using two *implicit* rules and then explicitly mentioning, as the proof-step, that ‘x = z (mod p)’.

The user is asked to put the proof-steps in the right order.

If you are looking for a simple challenge, consider making p a concrete number and dealing with constants as well as variables, calculating whatever you can calculate.

If you are looking for a hard challenge, consider finding a way to ensure that certain proof steps are guaranteed to be used in the proof (note that filtering out the proofs that use them won’t work from an efficiency point of view)

Meta: Expressions with binders*

Many of the CS30 topics require bound variables. These include: product, sum, forall, exists, a union over sets of sets, and set-builder notation.

In all of these assignments, I needed to work around that, see the set-builder notation's "extreme challenge" assignment earlier in this list of proposed final assignments.

In this assignment, you generate a proof based on operations that can bind variables. Note that this changes the syntax of rules.

As a convention, I'll use a . after every variable binding. For example: \forall x. [x \vee \not x] would be true in predicate logic. In the following expression:
\sum_s^e [x. f(x) + y]
f stands for an *expression* (rather than a function) that may be substituted even if it contains 'x'. The expression y, as before, still stands for an expression, but in this case it is an expression that may not contain x. The expressions s and e occur outside of the 'x' binding. Under this convention:
\sum_s^e x. [f(x) + y] = (\sum x. [f(x)]) + y * (e - s)
(Note that (e - s) counts how many elements are in the sum over the interval [s,e) or [s,e-1])

Here's a set of rules you can test your code on:
\sum_a^c x. [f(x) + g(x)] = \sum_a^b x. [f(x)] + \sum_a^b x. [g(x)]
\sum_s^e x. [y] = y * (e - s)

Try it out on the expression: \sum_a^b x. [\sum_c^d y. [x+y]]
Also, this should not rewrite using the above rules: \sum_a^b x. [\sum_x^d y. [y]]

Make a proof-of-concept exercise by randomly generating expressions.