Introduction to Computer Algebra

Due: 5/13/20

While we've learned how to do a lot in this class, we still haven't learned how to cheat at math homework. Specifically, it'd be nice if we could get our computer to do our algebra homework. This is a very important skill. Who knows, you may be somewhere where you don't have a calculator or phone, but still have your laptop for some reason. You'll need it to reduce all of those wild polynomials.

structure

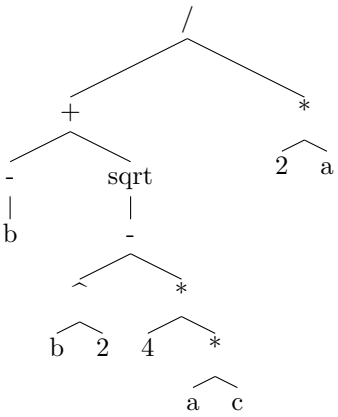
By this point, we're all used to thinking about algebra as manipulating strings of characters. $x^2 + 2x + 1$ can be reduced to $(x + 1)^2$. While this works fine for humans, this is not a good representation of algebraic expressions for computers. Seriously, try to write a reducing function

```
String reduce(String expr)
{
    ...
}
cout << reduce("x^2 + 2x + 1")
```

You might be able to get it to work in some cases, but it should also be able to handle `"x^2+2x+1"`, `"x^ 2 + 2 x + 1"`, and `"x^ 2+2 x+1"` When really this shouldn't matter.

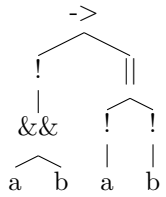
It turns out that trees make a good structure for representing expressions. If we have an expression $a + b$, we represent that as the tree:

If we have a more complicated expression like `( -b + sqrt(b^2 - 4*a*c)) / (2*a)` we can represent that with the tree:



The rules for constructing this tree follow the order of opoerations (or precidence) for algebra.

These trees are called *terms* (or ASTs in programming languages), and they will be the basis for this assignment, so it's worth understanding them. It turns out that we can make terms for other algebras. For example, take the boolean algebra expression `!(a && b) -> (!a || !b)`:



Now would be a good time to test your understanding.
Write down terms for the following expressions.

- `a + b + c`
- `a + b * c`
- `!!!!a`
- `!a && !b -> !(a || b)`

theory

Now that we've seen what a term looks like, we can mathematically define them. This may seem unnecessary, but if we're careful with our definition, it'll actually make it easier to write the code.

First, what is a term? we'll when we really look at it, there can be 3 types of nodes in our tree.

- a variable: x has no children.
- a literal: 4 has no children.

- a function: $sqrt, *$ has one or more children.
  We write f/n to say that function f has arity n.

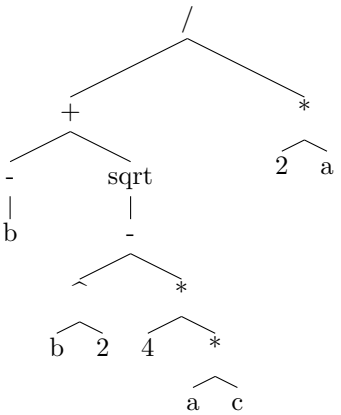
These are the only possibilities.

We can define a term inductively. A term t is either a variable v, a literal l, or a function $f/n(t_1, t_2, \ldots t_n)$ where $t_1, t_2 \ldots t_n$ are all terms.

as an example: `3 * x` is really the term `*/2(3, x)`. where `*` is a function with arity 2, `3` is a literal, and `x` is a variable. Since `3` and `x` are both terms, `*/2(3, x)` is as well.

If we want to talk about a piece of a term, we need to identify where it is in the term. Consider the quadratic formula again:

```
( -b + sqrt(b^2 - 4*a*c)) / (2*a)
```



The discriminant is the part under the square root. The *path* to a node in a term is a list of integers. We start at the root, and each integer tells us what branch to take. Formally, given the term $f(t_1, t_2, \ldots t_n)$ where x is in t_i, The path from the root to x is i followed by the path from t_i to x.

In our example the path to the discriminant is $[1, 2, 1]$, because we took the first branch at the /, the second branch at the +, and the first branch at the *sqrt*. We write this as $t|_{[1,2,1]}$. Yeah, that notation kind of sucks, but its standard.

Now that we have terms, we need to deal with variables. One of the most fundamental computations in algebra is substituting a variable with a value. It so fundamental that you probably never even thought of it as a computation. We'll capture this with a function. A *substitution* $\sigma : Var \to Term$ takes a variable, and returns a term. If we have $x + x = 4$, then we might find the substitution $\sigma(x) = 2$, However if we have $x + y = 4$, then we might have the substitution:
$\sigma_1(x) = 1$
$\sigma_1(y) = 3$
or we might have:
$\sigma_2(x) = 2$
$\sigma_2(y) = 2$
This notation gets long winded pretty quickly, so I'll use $\sigma_1 = \{x \mapsto 1, y \mapsto 3\}$

as a shorthand. We can also extend substitution to terms. $\sigma_1(x+y=4)$ means we substitute all variables in the term, which gives us $1+3=4$ Two terms, t_1 and t_2, *unify* if there exists some substitution u such that $u(t_1) = u(t_2)$.

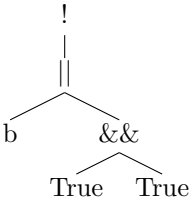
Finally, we need some idea of reducing an expression. A reduction rule $l \to r$ is just a pair of terms. a term t can be reduced by the rule $l \to r$ if there is a path p in t such that $t|_p$ unifies with l for some unifier u. to reduce the term, we replace $t|_p$ with $u(r)$. The way to think about this is that if a subterm of t matches the left hand side, then we can replace it with the right hand side.

As an example, we can use the absorption law from boolean algebra

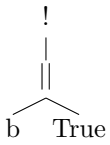
```
a && a -> a
```

and the term

```
!(b || (True && True))
```



We need to find 2 things. A path p and a unifier u so that $u(t|_p) = u(l)$ Our path is $[1, 2]$, and our unifier is $u = a \mapsto True$ Now, we just replace $t|_p$ with $u(r)$ (which is $True$).



And that's it. You've just done your first reduction.

Translating to Code

Ok, so this is where things get interesting. We'll need to represent everything we defined mathematically. Actually this was the purpose. Our definition will guide our implementation.

Remember our definition for a term. Variables, functions, and literals are all terms. Well, this sounds like a good place for inheritance. But, since I don't know what type of literals I'll be using, they should be templated.

```
template<typename T>
class term { ...  }

template<typename T>
class variable : public term<T>  { ... }

template<typename T>
class literal : public term<T>  { ... }
```

```
template<typename T>
class function : public term<T>  { ... }
```

We can represent a path as `vector<int>`. Remember all of our paths are 1-indexed.

I'm going to provide a `Sub` class for substitutions.

```
template<typename T>
class Sub
{
    term<T>& operator()(std::string s) const;

    bool contains(string s) const;

    void extend(string s, term_ptr<T> t);
}
```

If we have a `Sub sigma`:
`sigma("x")` will return the term that `x` maps to
`sigma.contains("x")` returns if the variable `x` has been added to the substitution.
`sigma.extend("x", t)` will add `x` to the substitution with term `t`.

Once we have these pieces, we're almost there.

For this assignment You need to provide definitions for the 4 classes defined above. They should be

- They should be STL complaint (with forward iterators)
- They should have copy constructors/assignment operators
- you need to overload `operator<<(ostream&, const term<T>&)`
- we need a rewrite function
  `term_ptr<T> rewrite(term_ptr<T> t, term<T>& rhs, vector<int> path, const Sub<T>& sigma)`

If you're doing the advanced assignment you need.

- Move constructors/assignment operators
- a unify function the computes a substitution
  `bool unify(const term<T>& t1, const term<T>& t2, Sub<T>& sigma)`
  If unify returns true, then sigma(t1) and sigma(t2) should be the same.
- a function to reduce a term using a list of rules.
  `term_ptr<T> reduce(term_ptr<T> t, const std::vector<rule<T>>& rules)`

Important points to remember.

- `term_ptr<T>` is just an alias for `shared_ptr<term<T>>`
- inheritance can really help you here. Think about how you want to set this up.
- You shouldn't be using anything resembling RTTI.
- Remember, never make a new `shared_ptr` out of an existing pointer.