Unification

1 Unification

While we have discussed variable usage at length, and variables have been used in each subsequent section, a full discussion of how variables get values has never been provided. These details are important for understanding how Prolog works overall, particularly in the context of "backwards execution", as previously described. We shed some insight on exactly how variables get values in this handout

In Prolog, variables are assigned values via *unification*. Unification is effectively mathematical equality (=), but in a way that can be implemented within the Prolog engine. In fact, `=` is used to tell the Prolog engine to unify two values, just as we would say that two values should be equal in math. Like mathematical equality, unification operates in either direction. For example, consider the following two queries:

```
1  ?- X = 1.
2  X = 1.
3  ?- 1 = X.
4  X = 1.
```

Both of these queries result in the assignment of `1` to `X`, and they fundamentally work in the exact same way. This is decidedly **unlike** the assignment operator from other languages, wherein the variable must be on the left and the value must be on the right. This is somewhat obnoxious because other languages still use the syntactic `=` for assignment, even though assignment bears little connection to its mathematical meaning. In contrast, Prolog's unification operator works just like mathematical =, and Prolog lacks assignment.

> ***Sidenote:*** Strictly speaking, there *is* a way to perform true assignment in Prolog, but not with the `=` operator. We may get into such features later in the course.

Also like mathematical = and unlike assignment, once a variable has a value, the value of the variable can never change. For example, the following query fails:

```
1  ?- X = 1, X = 2.
2  false.
```

At the point of `X = 1` in the above query, Prolog stores that the value of variable `X` is `1`. When `X = 2` is encountered, this ultimately triggers failure (leading the engine to report `false` above), as `1` and `2` are not equal to each other.

The related query below does, however, succeed:

```
1  ?- X = 1, X = 1.
2  X = 1.
```

At the first use of `X = 1`, the value of variable `X` is stored to be `1`. At the second use of `X = 1`, we simply check to see if the stored value of `X` is `1`. In this case, the stored value of `X` is equal to `1`, and so the query succeeds with the information that the value of `X` is `1`.

This process is easy enough to follow when we deal with exactly one variable and one value. However, part of the power of Prolog is that we can reason about many variables and values. For example, consider the following query:

```
1  ?- X = Y, X = 1, Y = 2.
2  false.
```

The above query fails because we first added a constraint that `X` and `Y` should be the same value, and then we tried to give `X` and `Y` separate values (namely `1` and `2`, respectively). If we had never issued the `X = Y` part in the query above, this query would have succeeded. Such a modified query can be seen below, which succeeds:

```
1 ?- X = 1, Y = 2.
2 X = 1, Y = 2.
```

What the above two queries show is that somehow we need to record if two variables hold the same value, **even if** we don't know exactly what that value is yet. This was the case for the `X = Y` portion above, which recorded that variables `X` and `Y` should hold the same value, even though values had not been given to `X` and `Y`.

This sort of recording can be most easily understood in terms of *equivalence classes*. Roughly, an equivalence class is a set of elements which are all declared to be equal. Initially, all values and variables are in their own separate equivalence classes. For example, let's say that we have variables `X` and `Y`, along with the value `1`. Initially, there are three equivalence classes: one containing `X`, a second containing `Y`, and a third containing `1`. If we issue a unification `X = Y`, this effectively states to *merge* the equivalence classes for `X` and `Y`. This merge leaves us with two equivalence classes: one containing both `X` and `Y` (hereinafter named $X \cup Y$), and a second containing the value `1`. If we then issue the unification `X = 1`, we will then take the equivalence class containing `X` (namely $X \cup Y$) and merge it with the equivalence class which just contains `1`. In this example, the end result is a single equivalence class holding variables `X` and `Y`, along with the value `1`. The semantic meaning of this is that variables `X` and `Y` share the same value, and that value is `1`. For the rest of the discussion, we will call this equivalence class $X \cup Y \cup 1$.

Care must be taken when merging equivalence classes. For example, if we take the equivalence class $X \cup Y \cup 1$ and attempt to merge it with an equivalence class containing `2`, this should fail: `1` does not equal `2`, and so we can never merge equivalence classes containing different values. In Prolog, such faulty attempts to merge (i.e., attempts to unify values which do not unify) lead to failure. This failure will either cause the entire query to fail, or at least go back to whatever the last choice was and (hopefully) try to unify with a different value.

In the last assignment, you will implement this described unification operation. At that point, we will discuss all the nitty gritty details involved in a basic implementation of this operation. For now, this explanation works to understand the basics of unification, though it's probably not enough information to go on to implement the operation.