THREE: PATTERN MATCHING

When you run a task, also known as *calling* the task, the system searches for a method that **matches** the parameters of the call. In particular, it matches the parameters of the call to the part of the method before the colon, known as its **head**. The head of the method:

```
    SomeTask 1 ?x: Bla bla bla
```

is the part that says "`SomeTask 1 ?x:`". It's the part that specifies what parameters this method is appropriate for. Parameters can either be variables, such as `?x`, that match anything or specific values that such as `1`, `john`, or `"Some quoted text"`. Things that aren't variables are called **constants**. Constants can only match to themselves: 1 matches 1, but not 2; `john` matches `john`, but not 1 or 2. Here are some examples:

a) **`SomeTask 1 2:`** `bla bla bla`
   Will only match a call with 1 and 2 for the parameters.
b) **`SomeTask 1 ?x:`** `bla bla bla`
   Will match any call that has 1 for the first parameter, since the `?x` can match anything. In addition, `?x` will get set to whatever the second parameter is.
c) **`SomeTask ?x 1:`** `bla bla bla`
   Will match any call that has 1 for the *second* parameter, and `?x` will be set to the second parameter.
d) **`SomeTask ?x ?y:`** `bla bla bla`
   Will match any call at all. `?x` will be set to the first parameter, and `?y` the second.
e) **`SomeTask ?x ?x:`** `bla bla bla`
   Will match any call at all in which the two parameters are *the same*. `?x` will be set to that shared value.

The last example here brings up an important point: if a variable appears more than once, **it must match the same value each time**.

Let's look at which methods each of the following calls would match. To save space, I've changed the name of the task from `SomeTask` to just `Task`:

	`Task 1 2:`	**`Task 1 ?x:`**	**`Task ?x 1:`**	**`Task ?x ?y:`**	**`Task ?x ?x:`**
`[Task 1 1]`	No	Yes: ?x=1	Yes: ?x=1	Yes: ?x=1, ?y=1	Yes: ?x=1
`[Task 1 2]`	Yes	Yes: ?x=2	No	Yes: ?x=1, ?y=2	No
`[Task 1 3]`	No	Yes: ?x=3	No	Yes: ?x=1, ?y=3	No
`[Task 2 1]`	No	No	Yes: ?x=2	Yes: ?x=2, ?y=1	No
`[Task 2 2]`	No	No	No	Yes: ?x=2, ?y=2	Yes: ?x=2
`[Task 2 3]`	No	No	No	Yes: ?x=2, ?y=3	No

MATCHING VARIABLES WITH OTHER VARIABLES

In the table above, we only included variables in the method heads, not in the calls. But you can include a variable in a call too. This behaves differently, depending on whether the system has already matched that variable to a value. If we say `[Task ?a 2]` but the system has already matched `?a` to 1, then we're really just doing `[Task 1`

2], and we can use the table above. But **if the system has never matched ?a** to anything, then ?a can be matched freely. That means three things:

- Specifying the variable in the call doesn't restrict the methods that can be used, since it will match anything
- The variable in the call *can be given a value by the method being called*, as a result of the matching.
- The matching process can result in the system deciding that two variables must have the same value, without yet having a value for either (see below).

Let's look at what does and doesn't match when we put variables in the call:

	Task 1 2:	Task 1 ?x:	Task ?x 1:	Task ?x ?y:	Task ?x ?x:
[Task 1 ?a]	Yes; ?a=2	Yes: ?a=?x	Yes: ?x=1, ?a=2	Yes: ?x=1, ?y=?a	Yes: ?x=1, ?a=1
[Task ?a 1]	No	Yes: ?x=2, ?a=1	Yes; ?a=?x	Yes: ?x=?a, ?y=1	Yes: ?x=1, ?a=1
[Task ?a ?b]	Yes; ?a=1,?b=2	Yes: ?a=1,?b=?x	Yes; ?a=?x,?b=1	Yes: ?x=?a,?y=?b	Yes: ?a=?b=?x
[Task ?a ?a]	No	Yes: ?a=?x=1	Yes: ?a=?x=1	Yes: ?x=?y=?a	Yes: ?x=?a

Notice that for many of these, specifically the ones in green, the system is able to match the variables, but doesn't come away from it knowing what the values of some of the variables are. It only knows that certain variables have have the same values as one another. It remembers that, so that if it matches either variable to a value in the future, then it will know that *both* variables must have that value.

FAILED MATCHES

Finally, it should be pointed out that if the system gets partway through a match, but then fails, for example because the first parameters match, but not the second, then any changes to the variables made during the match get undone. For example, when matching [Task ?a 1] to [Task 1 2], above, the system will tentatively set ?a to 1 when matching the first parameter. But when the second parameters fail to match, it restores ?a to its original, "unset" state.

NAMING OF VARIABLES

In these examples, I've been at pains to make sure that the variables that appear in the calls have different names from the variables that appear in the methods, so as to prevent confusion. So this is a good time to mention that:

- Variables with the name in the same method are interpreted as meaning the same variable (we knew this already).
- However, variables in different methods with the same name are treated as different variables that just happen to have the same name. The system won't confuse them with one another.
- If a task is called more than once, each call has its own set of variables. So calling a task doesn't permanently set any of its variables.

ARE YOU CONFUSED YET?

I know this is confusing. It takes some getting used to. But remember that this is just describing something you'd already understood at least the simple cases of. We've just gone into more detail to talk about how things are working under the hood. Knowing this will help you when we look at some fancier techniques. But most of the time as a programmer, you don't have to think down at this level of detail.

FOR PROGRAMMERS

If you've programmed before, this may have been a kind of kick in the head. Variables work very differently than they do in Python. Whereas in Python, variables get their values either by being parameters that are filled in during a call, or by being updated with assignment statements like x=x+1. In *Step*, variables more or less only get values through pattern matching. Moreover, once a variable has a value, you can't update it. This turns out to be very useful for certain kinds of programming.

Later, we will introduce other kinds of variables you can update with assignment statements, so you will be able to write x=x+1. But the vast majority of the code you write will work through pattern matching.

FOR CS MAJORS (OPTIONAL)

In AI research and programming language research, the pattern matching technique used here is known as **unification** and the algorithm used is called the **unification algorithm**. If you want to know how it works, here's a sketch.

A variable basically looks like an object that can either point at nothing (it has no value), an object (that's its value), or another variable that it's supposed to be equal to, which we'll call its substitute. When we work with a variable, we always check to see if it has a substitute. If so, we use the substitute instead. It may have a substitute itself, but we eventually reach some variable that doesn't have a substitute. We'll call that the "final substitute." Nearly all the time, when we work really work with its final substitute. The "real" value of the variable, if any, is the value of its final substitute.

```
class Variable {
   // The value of the variable, if any
   public object Value;
   // If non-null, the substitute actually holds this variable's value
   public Variable Substitute;

   // Follow the chain of substitutes to find a variable that isn't
   // substituted
   public object FinalSubstitute() {
      for (var v = this; v.substitute != null; v = v.substitute);
      return v;
   }

   // Mark that this variable is equal to o.
   // This should only ever be called when both Value and Substitute are null.
   public void SetEqual(object o) {
     if (o is Variable v)
        Substitute = v;
     else
        Value = o;

   // The value of the variable, if any.
   public object RealValue() => FinalSubstitute().Value;
}
```

We also define a help function, `Finalize`, that takes an object and returns it if it isn't a variable. If it's a variable, it finds its final substitute, and if that final substitute has a value, returns that. Otherwise, it returns the final substitute:

```
public object Finalize(object o) {
  if (o is Variable v) {
     v= v.FinalSubstitute();
     if (v.Value != null)
        return v.Value;
     else return v;
  }
  else return o;
}
```

Matching two values then looks like this:

```
public bool Match(object a, object b) {
   a = Finalize(a);
   b = Finalize(b);
   if (a is Variable finalA) {
      finalA.SetEqual(b);
      return true;
   } else if (b is Variable finalB) {
      finalB.SetEqual(a);
      return true;
   }
   return a == b;
}

public bool MatchArray(object[] a, object[] b) {
  if (a.Length != b.Length) return false;
  for (var i = 0; i< a.Length; i++)
     if (!Match(a[i], b[i])) return false;
  return true;
}
```

This is pretty much the algorithm. It correctly tells you if two values or arrays of values match one another, and handles any equalities between variables it finds along the way. The thing it doesn't handle is undoing the modifications to the variables if you get partway through and realize they don't match. We'll talk about how to handle that later.