FOUR: PREDICATES AND QUERIES

Now we'll to look at a set of techniques that will be very useful for text generation. However, it's easiest to explain them using examples that aren't directly related to story or text generation. So for most of this, we'll ignore text generation. Don't worry; we'll get back to stories shortly.

HOW TO READ THIS

This is probably the most technically subtle of the topics we'll look at. In particular, the section on **Backtracking** is hard to get the details of the first time around. Read through it, but don't feel you need to follow it in detail at this point. It's there to give you a sense of what's happening under the hood, but most of the time you don't have to think about what's happening underneath. I want you to have a general sense of how the computer is running the code, but learning the details can come later.

METHODS THAT DON'T PRINT ANYTHING

Let's start by observing that methods can do things besides printing text. In particular, they can succeed or fail, and when they do succeed, they can give values to variables through the matching process we talked about in the last section.

So now let's think about tasks that don't print any text. For example, if we have a method that looks like this:

```
MyTask:
[end]
```

Then that method runs without print anything. This might be useful, for example, if you want to have a task that prints descriptive adjectives for some other word, and you want one of the options (one of the methods) to be not to print anything. These textless methods are surprisingly common. Enough so that there's a shorthand to keep you from having to put in the colon and the [end]. Instead, you can just put a period at the end:

```
MyTask.
```

This means exactly the same thing as the previous version, it's just shorter and hopefully easier to read.

Here's a more sophisticated example. Suppose we're writing dialog for a scene where two characters talk. We might write a task, `Greet`, that takes two parameters, the speaker (who's speaking) and the addressee (who's being greeted), and the greeting that speaker would say to that addressee, whatever it might be. Here's an example of how we might write that:[1]

```
Greet bill john: Hello, boyfriend!
Greet bill pat.
Greet ?_person1 ?_person2: Hey.
```

[1] You may wonder why we named the variable `?_person1` rather than just `?person1`. It's because it's used only once. That's often a mistake in the code, and so Step prints warning for single-use variable unless they start with "?_" or are just named "?". Using those names tells Step, you mean them only to be used once.

This says that people greet one another by saying "Hey." Bill does too, unless they're talking to John or Pat. He greets John by saying "Hello, boyfriend!" and greets Pat with stony silence because they had a falling out last week. The computer tries them in the order listed (because it's not tagged with `[randomly]`), so the last line acts as a default that the previous lines override.

So the "`Greet bill pat.`" method doesn't print any text. You might be tempted to say this method doesn't do anything, but that's not quite true: it signals that the task has been successful and we don't have to try any more methods. In this case, that means it prevents the system from going on to the next method and printing "Hey."

Let's write down what each of the lines of Greet mean:

Line	**Meaning**
`Greet bill john:` `Hello, boyfriend!`	If Bill is greeting John, say "Hello, boyfriend!"
`Greet bill pat.`	Otherwise, if Bill is greeting Pat, print nothing
`Greet ?_person1 ?_person2:` `Hey.`	Otherwise, if anyone is greeting anyone, print "Hey."

TASKS THAT NEVER PRINT ANYTHING

So that's an example of wanting a method that doesn't print text. Not only are those common, it's common to have tasks none of whose methods print anything. Let's look at an example:

```
[predicate]
Dog rover.
Dog fluffy.
[predicate]
Cat boots.
Cat ulysses.
```

We'll talk about the `[predicate]` annotation in a minute, but what does the rest do? Why would you want to write code like this? The answer is that this is a little database of pets: Rover and Fluffy are dogs, Boots and Ulysses are cats. If you call `[Dog rover]`, it doesn't print anything, but the task succeeds. On the other hand if you call `[Cat rover]`, the task fails because there's no "`Cat rover.`" method. So even though it doesn't print anything, we can use it to test whether a pet is a dog or cat. That means we can use it to control other tasks that do print things:

```
Talk ?pet: [Dog ?pet] Woof!
Talk ?pet: [Cat ?pet] Meow!
```

What happens if we say `[Talk rover]`?

- It tries the first method, setting `?pet` to `rover`
- Then it runs `[Dog ?pet]`
- Since `?pet=rover`, it's really running `[Dog rover]`
- That matches the first method for Dog, and so it succeeds, without printing anything
- `Talk`'s method then prints "Woof!"

What happens if we say `[Talk boots]`?

- It tries the first method, setting `?pet` to `boots`
- Then it runs `[Dog ?pet]`
- Since `?pet` is `boots`, it's really running `[Dog boots]`
- That doesn't match any method for Dog, and so it fails
- So `Talk`'s first method fails
- Next it tries `Talk`'s second method, again setting `?pet` to `boots`.
- Now it runs `[Cat ?pet]`, which is to say `[Cat boots]`, and that works
- So it prints "Meow!"

Once again, let's go over this code line by line and describe in English what the programmer was trying to communicate to the computer:

Line	Meaning
`Dog rover.`	Rover's a dog!
`Dog fluffy.`	So is fluffy
`Cat boots.`	Boots is a cat
`Cat ulysses.`	So is Ulysses
`Talk ?pet:` `[Dog ?pet] Woof!`	If a pet is a dog, then they say "Woof!"
`Talk ?pet:` `[Cat ?pet] Meow!`	If a pet is a cat, then they say "Meow!"

So the Dog and `Cat` tasks are ways of asking is a character is a dog or cat, and the code for them is essentially a little database of dogs and cats. You query the database by saying `[Dog` *something*`]` or `[Cat` *something*`]` and it succeeds if *something* is one of the things it knows to be a dog or cat, respectively, otherwise it fails.

WILDCARD PARAMETERS

Now let's ask what happens if we run this command:

```
[Dog ?who]
```

That is, what happens if we call Dog, but we don't pass it a name like `rover` as a parameter. Instead, we pass in a variable, `?who`. In that case, the variable acts as a *wildcard*: it will match any value in the method. So when it goes to run the command, it starts with the "`Dog rover.`" method, as usual. That method matches even though the parameter value we specified wasn't `rover`, because we gave it a variable and the variable can match anything.

So the task succeeds. But here's the cool thing. The task doesn't just succeed, it reports back the new value for `?who`. So if you type the line above at Step, it replies:

```
?who = rover
```

So in other words, calling `[Dog rover]` effectively asks the **yes-or-no question**, "Is Rover a dog?", and by succeeding, the Dog task answers "yes". But calling `[Dog ?who]` effectively asks the **wh-question** "who is a dog?".[2] In this case, it replies that Rover is a dog, by succeeding with `?who = rover`. In this case, it will always reply that Rover is a dog because that line comes first. But if we add a `[randomly]` annotation:

[2] In linguistics, a wh-question is a who, what, why, where, or how question. That is, it's a question that asks for more than a yes/no answer.

```
[predicate] [randomly]
Dog rover.
Dog fluffy.
[predicate] [randomly]
Cat boots.
Cat ulysses.
```

We turn it into a random dog and cat generator. Each time you do a query like `[Dog ?x]`, we get back a randomly chosen dog in the variable `?x`.

PREDICATES

So now let's talk about the `[predicate]` annotation that we'd been avoiding talking about. The term *predicate* comes from the Latin *praedicātum*, meaning "thing said about some subject." It's the source of the subject/predicate distinction in linguistics, and the legal notion of the predicate for a criminal prosecution. The usage of the term in computing comes from logic, where it means a true-or-false property of something or relationship between a number of somethings: a predicate is either true or false for a given set of parameters. In *Step*, that ends up meaning a predicate is just a task that either succeeds (answering true) or fails (answering false), but it doesn't print anything. In a sense, a predicate is a task you use to answer a question.

On a technical level, what the `[predicate]` annotation tells *Step* is that it's okay if you call the task and it fails.[3] Under normal circumstances, *Step* will assume that a task failing completely is an indication of a bug in the code, and so it will print an error message. But if a task is tagged as being a `[predicate]`, it won't get upset.

So predicates are tasks that don't generate output, and are just used to record and test information. A predicate takes some parameters, and either succeeds (possibly giving values to variables in the process) or fails. Here are some questions you can ask of our Dog and `Cat` predicates:

Query	**Meaning**
`[Dog fluffy]`	Is Fluffy a dog? (answer: yes)
`[Cat fluffy]`	Is Fluffy a cat? (answer: no)
`[Dog ?x]`	Who is a dog? (put the answer in ?x)
`[Cat ?who]`	Who is a cat? (put the answer in ?who)

RELATIONS

As with any other tasks, predicates can have as many parameters as you like. When a predicate takes multiple parameters, it's used to express relationships between the parameters. For example, we could add a predicate, `HumanOf` that expresses who the human companions of the different animals are:

[3] A note to would-be power-users. Technically, `[predicate]` means both that a call can fail, and also that a call can potentially succeed in multiply ways. We'll talk about this more later, but note for example that the query `[Dog ?x]` actually has two possible solutions: `?x` = `rover` and `?x` = `fluffy`. Marking a task with `[predicate]` not only says it's okay for it to fail, but also that it's okay to consider the `fluffy` solution if the `rover` one doesn't end up being the dog we're looking for.

```
[predicate]
HumanOf rover jane.
HumanOf fluffy bill.
HumanOf boots chris.
HumanOf ulysses jane.
```

Now we can write queries about the relationships between humans and pets:

Query	Meaning
`[HumanOf fluffy chris]`	Is Fluffy's human Chris? (answer: no)
`[HumanOf boots ?who]`	Who is Boots' human? (answer: Chris)
`[HumanOf ?who bill]`	Who is Bill's pet? (answer: Fluffy)
`[HumanOf ?pet ?human]`	Tell me a `?pet` and their `?human`, I don't care which

We can use predicates to enter whatever information we like about pets or cars or Chinese food, and then query it however we like.

CHAINING QUERIES

We can also define predicates in terms of other predicates. For example:

```
[predicate]
HasCat ?human: [HumanOf ?pet ?human] [Cat ?pet]
```

Here, the intention is to define a predicate, `HasCat`, that is true (i.e. that succeeds) when its parameter is a human who has a cat. So if our code works, the query `[HasCat chris]` would mean "does Chris have a cat?" and should succeed if the database says Chris has a cat, and fail otherwise. Let's look at how that command would run:

- First, it would match `[HasCat chris]` with `HasCat`'s method. They match by making `?human` = `chris`.
- Now, it tries to run the rest of the method, subject to the stipulation that `?human` has to be `chris`.
- It tries to run `[HumanOf ?pet ?human]`. Since `?human` = `chris`, it's really running `[HumanOf ?pet chris]`, i.e. "who is Chris' pet?"
- That matches against the method "`HumanOf boots chris.`" So the query succeeds, and in the process gives us the solution, `?pet` = `boots`.
- Now we move on to run `[Cat ?pet]`, but since `?pet` = `boots`, we're really running `[Cat boots]`.
- We happen to have a method that says "`Cat boots.`" so `[Cat boots]` succeeds.
- That means everything in `HasCat`'s method has succeeded, so the original call `[HasCat chris]` succeeds, meaning our answer is "yes, Chris has a cat."

BACKTRACKING

Now what happens if we say `[HasCat jane]`? The short answer is that the system runs the method:

```
HasCat ?human: [HumanOf ?pet ?human] [Cat ?pet]
```

with `?human` = `jane` and tries to find a `?pet` of `jane` that is also a cat. There is one, Ulysses, and so it succeeds. Most of the time, you don't have to think any deeper than this. We asked if Jane had a cat, it checked, and it found one. But it's worth talking about how exactly it goes about finding it. This is how it works.

- As before, it matches [HasCat jane] with the HasCat's method. They match by making ?human = jane.
- So as before, it tries to run the rest of the method, although now ?human is jane rather than chris.
- As before, it tries to run [HumanOf ?pet ?human]. Since ?human = jane, it's really running [HumanOf ?pet jane], i.e. "who is Jane's pet?"
- As before, that matches a method, but in this case the first method that matches is "HumanOf rover jane." Again, the query succeeds. In the process, it tells us that ?pet = rover.
- As before, we run [Cat ?pet], but since ?pet = rover, we're really running [Cat rover].
- This time, that doesn't work; [Cat rover] doesn't match any method for Cat, so it fails.

At this point, the system knows that Jane has a pet named Rover, but that Rover isn't a cat. That's not useful. As a result, the system **backtracks** to see if Jane has another pet:

- The system abandons the [Cat rover] query and revisits the query before it: [HumanOf ?pet jane].
- That means it also abandons thinking that ?pet = rover, since that was a result from running HumanOf before
- It picks up where it left off scanning through the methods of HumanOf, looking for a method that matches [HumanOf ?pet jane]
- It finds that the last method, "HumanOf ulysses jane.", matches. So HumanOf succeeds (again), but this time yielding that ?pet = ulysses.
- Now we move on to run [Cat ?pet], but since ?pet = ulysses, we're really running [Cat ulysses].
- We happen to have a method that says "Cat ulysses." so [Cat ulysses] succeeds.
- This means we made it all the way through HasCat's method, and so HasCat succeeds.

WILDCARDS, REVISITED

Now what happens if we run [HasCat ?who], that is, "who has a cat?", or to look at it from the computer's standpoint, "find me a value of ?who for which ?who has a cat". Again, that the system runs the method:

```
HasCat ?human: [HumanOf ?pet ?human] [Cat ?pet]
```

But this time all the matching process tells it is that ?who = ?human, meaning it doesn't know the values of either variable, it just knows they have to have the same value. Then it does something pretty similar to what it did before:

- As usual, it runs [HumansOf ?pet ?human], although ?human = ?who so we can also think if it as running [HumansOf ?pet ?who].
- That call matches any of the methods of HumansOf, since neither variable has a value. So it just tries them one at a time, in order.
- When we match to the first method, we get ?pet = rover, ?human = ?jane. Since ?human is the same as ?who, that means ?who is also the same as jane, but that's not super important at this point.
- Now we run [Cat rover], which we already saw fails.
- So now the system tries successive methods of HumansOf until it finds one that works.
- The second method gives us ?pet = fluffy, ?human = ?who = bill, but [Cat fluffy] also fails, so that doesn't work either.
- The third method gives us ?pet = boots, ?human = ?who = chris.
- Now it tries running [Cat boots], which matches the method "Cat boots." So that succeeds.

- Success! We got to the end of the original method for `HasCat`!
- So `[HasCat ?who]` succeeds. Along the way, it learned that `?who` = `?human`, but it also learned that `?human` = `chris`. So the system prints out" `?who = chris`.

LOGIC PROGRAMMING

Hopefully the examples above convinced you that the definition:

```
[predicate]
HasCat ?human: [HumanOf ?pet ?human] [Cat ?pet]
```

Defines a predicate, `HasCat`, that is true when the database says its parameter has a cat and that, like other predicates, you can use it either to verify that a specific person has a cat, or by passing it a variable with no value, ask it to *find* a person who has a cat.

Because of the way the system operates, we can often pretend that our methods for predicates are not so much code as statements about our story world. The code:

```
[predicate]
Dog rover.
Dog fluffy.
Cat boots.
Cat ulysses.

[predicate]
HumanOf rover jane.
HumanOf fluffy bill.
HumanOf boots chris.
HumanOf ulysses jane.
```

Can be read as instructions for how to perform tasks, but also as **statements of fact**: Rover is a dog, Boots is a cat, Jane is Rover's human, etc. Moreover, we can read the definition:

```
[predicate]
HasCat ?human: [HumanOf ?pet ?human] [Cat ?pet]
```

As another statement of fact: *a human has a cat, if they have a pet and that pet is a cat*. Then we can think of running a query as a kind of logical reasoning. When we run `[HasCat chris]`, we're asking "does Chris have a cat?" The system (sort of) says "Chris has a cat if they have a pet who is a cat; they have a pet, Boots; and Boots is a cat; so therefore Chris has a cat."

This approach of trying to program by giving the system "facts" and letting the system "reason" about them is called **logic programming**, and there are people who advocate it as a general approach to programming. For this course, all you really know is that it's a good way of telling your story generator things about your characters (e.g. by putting facts about them into the "database"). You can then let your story generator ask questions about the characters by running queries. We'll look at examples of that in the next section.

SUMMARY

Here are the big lessons from this section.

- It’s useful to make tasks, called predicates, that never print anything; they just succeed or fail. When they succeed, they can return information to you by giving values to variables.
- Methods for predicates can often be read as statements of fact.
    - Methods of the form “`Predicate` *x y z*” can be read as “`[Predicate` *x y z*`]` is true (whatever *x*, *y*, and *z* are).”
    - Methods like “`Predicate` *x y z*`: [OtherPredicate` *x y*`] [YetAnotherPredicate` z`]`” can be read as “`[Predicate` *x y z*`]` is true **if** `[OtherPredicate` *x y*`]` is true and `[YetAnotherPredicate` *z*`]` is also true”
    - The system searches the different methods in the system for a set of methods that give it a consistent set of values for the variables that make it all work.
- Executing a predicate can be thought of as asking a question (querying)
- Execution works by a process called backtracking. But at least for simple code, you can often ignore the details and just trust that the system will find the right answer on its own.