Part 3: Predicates and databases

We looked before at how we can make a kind of little database of what pronouns the characters in our stories use:

**Reflexive “Mr. Boss”:** himself  
**Reflexive “Mary”:** herself  
**Reflexive ?:** themselves

This is a little bit of a pain, because there are actually several other kinds of pronouns besides the reflexive ones. There are the “normal” pronouns like “he”, “she”, and “them” but these have both a “subject” case (she/he/they) and an object case (them/her/him). There are also possessive adjectives (his/her/their). It’s a pain to write all of these down separately:

**SubjectivePronoun “Mr. Boss”:** he  
**SubjectivePronoun “Mary”:** she  
**SubjectivePronoun ?:** they

**ObjectivePronoun “Mr. Boss”:** him  
**ObjectivePronoun “Mary”:** her  
**ObjectivePronoun?:** them

**Possessive “Mr. Boss”:** his  
**Possessive “Mary”:** her  
**Possessive ?:** their

You can imagine that if you have a story with dozens of characters, this could be a problem. But we can simplify this considerably if we take into account the fact that most people use either all masculine pronouns, all feminine ones, or all epicine ones (they/them/their). It would be nice if we had a way of saying “Chris uses feminine pronouns” and not have to write each of them down.

We can do that by noting that nothing requires a task to generate output. If we just say:

**MasculinePronouns “Mr. Boss”:  
[end]  
FemininePronouns “Mary”:  
[end]**

We’re saying “if you call MasculinePronouns with Mr. Boss as a parameter, don’t print anything”. We’re also saying, via the lack of any other methods that if you call it with another parameter, it should fail. Now we can say:

**Reflexive ?who:** [MasculinePronouns ?who] himself  
**Reflexive ?who:** [FemininePronouns ?who] herself  
**Reflexive ?who:** themselves

Now, when you call Reflexive on “Mr. Boss”, the first method runs with ?who=”Mr. Boss”. It then calls [MasculinePronouns “Mr. Boss”], which succeeds but doesn’t print anything. So Reflexive continues and prints “himself”.

On the other hand, if you call Reflexive with “Mary” as its parameter, then the call to [MasculinePronouns “Mary”] will fail, causing the first method to fail. That’s okay. That’s why you have those other methods there. So when the first method fails, it just moves on to the next method, and that one will succeed because the call [FemininePronouns “Mary”] will succeed.

Tasks that don’t generate output, and are just used to record and test information are called **predicates**.

## IMPORTANT: How you’d *really* write it

So now, I have to admit that the code above won’t actually work. The reason is that normally in Step, a method can fail, but a task failing entirely (i.e. *all* its methods failing) is assumed to be a bug, and causes the program to stop. But you can override that by marking a task as being allowed to fail. The way we’d really write this is:

[predicate] **MasculinePronouns “Mr. Boss”.**

[predicate] **FemininePronouns “Mary”.**

The **[predicate]** annotation tells the system that it’s okay for it to fail.[[1]](#footnote-1) Like [randomly], you only have to tag one of the methods of a task with [predicate] and it will understand that the task is, in general, a predicate. Also, notice that we’ve replaced the colons at the ends of the methods with periods. That tells the system the method isn’t supposed to generate output, so you don’t have to write the colon or the [end] marker.

We can also make Reflexive a little more compactly:

**Reflexive [MasculinePronouns ?who]:** himself  
**Reflexive [FemininePronouns ?who]:** herself  
**Reflexive ?who:** themselves

When you write [Predicate ?variable] in the parameters of a method, the system knows the method only applies if the Predicate succeeds on the variable.

# Making a simple database

Now let’s look at how we can keep track of more information. Let’s suppose we’re generating a story that takes place at a restaurant. For example, it might start like this:

Jennie and Billy went to an Italian restaurant. Billy ordered spaghetti and Jennie ordered scampi.

If we want to switch up the food they order, we could do something like:

**Opening:**  
Jennie and Billy went to an Italian restaurant. Billy ordered [Food] and Jennie ordered [Food].  
**[end]**

[randomly]  
**Food:** spaghetti  
**Food:** scampi  
**Food:** carbonara

There are a few issues with this. One is that Billie and Jennie might both order the same thing, and the sentence:

Billy ordered scampi and Jennie ordered scampi.

Looks odd; you’d expect it to be “Jennie ordered scampi too” or “they both ordered scampi”. Another is that it’s only going to work if they’re at an *Italian* restaurant. If you start including non-Italian food in the list, then you need to make it’s not saying they’re at an Italian restaurant but somebody is ordering *pad thai* or that they’re at a Thai restaurant and somebody is order *huevos a la Mexicana*.[[2]](#footnote-2)

## Wildcard parameters

We can fix this problem by first having the system decide what cuisine to use and then pass that cuisine as a parameter to the Food task. To do this, we’re going to have a task, Cuisine, that picks a cuisine. We could do that like this:

**Opening:**Jennie and Billy went to a [Cuisine] restaurant. Billy ordered [Food] and Jennie ordered [Food].  
**[end]**

[randomly] **Cuisine:** italian  
**Cuisine:** thai

The problem with this is that there’s no way for the Food task to know what type of food was chosen by the Cuisine task. We can fix that like this:

**Opening:**[Cuisine ?c] Jennie and Billy went to a ?c restaurant.  
Billy ordered [Food ?c] and Jennie ordered [Food ?c].  
**[end]**

[randomly] **Cuisine** italian.  
**Cuisine** thai.

Notice that Cuisine and Food are each called with a variable and the same variable, as input. Moreover, that variable isn’t a parameter to Opening. What this means is “I don’t care what value the variable has; just that it needs to be the same in all those calls.”

[randomly]  
**Food italian:** spaghetti  
**Food italian:** scampi  
**Food italian:** carbonara  
**Food thai:** pad thai  
**Food thai:** pad see ew

The first subtask in Opening calls Cuisine and passes in a variable as a parameter. That’s a technique we haven’t seen before.

1. It’s actually saying more than that it’s okay for it to fail, but we’ll get to that later. For now, you can think of it as just saying it can fail. [↑](#footnote-ref-1)
2. In the interest of intellectual honesty, I should mention that hybrids do exsit. When I was growing up in Minneapolis, there was an actual Mexican-Chinese restaurant called *Que Pasa House of Chow*. But such things are rare enough that one doesn’t necessarily want to include them in one’s story generators, for fear that a seemingly minor decision from the standpoint of the generator will generate out-sized importance in the mind of the reader because of its perceived unlikeliness. [↑](#footnote-ref-2)