

FIVE: INFERENCE

At this point, we have enough tools to let the computer do something that we might call “reasoning.” We have predicates, which in *Step* are just tasks that don’t print things. That doesn’t mean they don’t do anything; it means they succeed or fail, and if they succeed, they potentially fill in values for variables passed to them that didn’t already have values. We think of calling a predicate as asking a question. For example, we might call `[Loves rover fluffy]`. If it succeeds, the answer is yes: Rover loves Fluffy. If it fails, then answer is no; poor Fluffy.

We can also ask “who/what” questions by passing in a variable and letting the predicate fill it in. If we run the command `[Loves rover ?x]`, we’re effectively asking “who does Rover love?” If it succeeds and fills in a value for `?x`, then we know Rover loves whoever was filled in for `?x`. Rover might love other characters too, but we at least know one of them.

RULES FOR REASONING

We’ve also seen how we can specify methods for predicates that are general rules about the truth of one predicate in terms of the truth of others. For example, if we already have information stored about the parents of characters, we don’t have to separately store information about sibling relationships, we could derive that from the parentage information:

```
[predicate]
Siblings ?a ?b: [Parent ?a ?parent] [Parent ?b ?parent]
```

As far as the computer is concerned, that’s just a rule that says “if trying to run `Siblings`, one way to do it is to first run `[Parent ?a ?parent]`, and if that succeeds, run `[Parent ?b ?parent]`”.

But another way of looking at it is that that can only work if `?a` and `?b` have the same parent, at least assuming `Parent` is defined so that `[Parent ?x ?y]` is true whenever `?x`’s parent is `?y`. So we could read this method as a rule that effectively says “people are siblings if they share a parent”.

This is a general technique we can use to write rules of the form “this is true if these things are also true.” If we want to say `A` is true if `B` is true, we write:

```
[predicate]
A: [B]
```

If we want to say `A` is true when `B`, `C`, and `D` are all true, then we say:

```
[predicate]
A: [B] [C] [D]
```

If we want to say “`A` is true when either `B` is true or `C` and `D` are both true”, we can write two methods:

```
[predicate]
A: [B]
A: [C] [D]
```

If we add parameters to the rules, everything needs to match up as it would for any of the other code we've seen.

REASONING ABOUT FAMILY RELATIONSHIPS

Let's return to our definition of siblinghood:

```
[predicate]
Siblings ?a ?b: [Parent ?a ?parent] [Parent ?b ?parent]
```

It's worth pointing out that under this definition, people are their own siblings. If you call `[Siblings fred fred]`, it will first run `[Parent fred ?parent]`, which will succeed, assuming Fred has a parent, and then it will run it again, so it will presumably succeed again. So it thinks Fred is his own sibling. That might work for your purposes or not. If you want to rule out self-siblinghood, you can just add an extra condition to the rule that requires `?a` and `?b` be different:

```
[predicate]
Siblings ?a ?b: [Parent ?a ?parent] [Parent ?b ?parent] [Not [= ?a ?b]]
```

Here, `=` just tests if its parameters are the same, and `Not` says that it should succeed if `=` fails, and fail if `=` succeeds. `Not` turns out to be complicated; we'll talk more about it later. But for this usage, it's pretty simple.

Now we can start to define other relationships:

```
[predicate]
Grandparent ?c ?g: [Parent ?c ?p] [Parent ?p ?g]
```

Which says that `?g` is a grandparent of grandchild `?c` if `?p` is a parent of `?c`, and `?g` is a parent of `?p`. Note that we're just choosing the convention that the grandparent is the second parameter. We could have chosen it to be the other way around; the computer doesn't care.

Or we can say:

```
[predicate]
Aunt ?c ?a: [Parent ?c ?p] [Siblings ?p ?a]
```

Which says that the aunt of a child is the sibling of a parent of a child. We haven't been paying attention to gender in these definitions, but since there isn't a good gender-neutral word for aunt-or-uncle, we'll just say aunt. If you want to track gender, then just add some predicates for `Male` and `Female`, and then add the predicate to the rule:

```
[predicate]
Aunt ?c ?a: [Parent ?c ?p] [Siblings ?p ?a] [Female ?a]
```

SELF-REFERENTIAL RULES

So far, so good. We can define the `Parent` relationship and then define most everything else in terms of that: grandparents and grandchildren, great grandparents, siblings, etc. But what about ancestors? An ancestor is your parent *or* grandparent *or* great grandparent *or* great, great grandparent, etc. We could write:

```
[predicate]
Ancestor ?c ?a: [Parent ?c ?a]
Ancestor ?c ?g: [Grandparent ?c ?g]
```

```
Ancestor ?c ?g: [Greatgrandparent ?c ?g]
Ancestor ?c ?g: [Greatgreatgrandparent ?c ?g]
```

And so on, forever. But forever is a lot of typing. What we can say instead is: an ancestor is either your parent, or an ancestor of your parent. That's just two rules that do the equivalent work of an infinite number of rules:

```
[predicate]
Ancestor ?c ?p: [Parent ?c ?p]
Ancestor ?c ?a: [Parent ?c ?p] [Ancestor ?p ?a]
```

In mathematics, logic, and computer science, this is called a *recursive* definition, and when the Ancestor rule turns around and calls Ancestor again, that's called *recursion*.

Recursive definitions have a fairly standard format. They generally start with a rule that doesn't recurse. This is called a *base case*. It notices when it can decide immediately without having to recurse. Then there's another rule that recurses, called unsurprisingly, the *recursive case*. **It's important to have the base case first**, and the recursive case second. And for the recursive rule, **you generally want to end with the recursive call**, not start with it.

Why is that? Because *Step* doesn't really understand logic. It's just mechanically running the tasks you tell it to. We're using our understanding of how it runs them to in some sense trick it to do logical inference. By putting the base case first, we're causing it to always check whether it really needs to recurse before recursing. If we flip the order, it will always recurse. That means the first call recurses, but then the recursive call recurses, and that call also recurses, on and on infinitely, without *Step* ever understanding that something's wrong.

So if you rearrange things, it's still conceptually correct from the standpoint of what it means to be an ancestor, but *Step* won't be able to work with it.

TAXONOMIES

Taxonomic relationships come up a lot in games. Dogs and cats are kinds of animals, Collie and Labrador are kinds of dogs. We can express those relationships using rules:

```
[predicate]
Animal ?a: [Dog ?a]
Animal ?a: [Cat ?a]
```

```
[predicate]
Dog ?d: [Labrador ?d]
Dog ?d: [Collie ?d]
```

```
[predicate]
Collie lassie.
[predicate]
Labrador bruce.
```

Now suppose we run `[Animal lassie]`, i.e. we ask "is Lassie an animal?" The system will start by running the first Animal method, which will check if Lassie is a dog. So then, it will run the first Dog method, which checks if Lassie is a Labrador. That fails, since Bruce is the only Labrador. So then the system backtracks and tries the next

Dog rule, which says to check if Lassie is a Collie. That succeeds, meaning it's verified Lassie is a Collie. That then means that the method it was running for Dog, which had called Collie, also succeeds, and so the system has verified Lassie is a dog. That then means that the method for Animal succeeds, and so it's verified that Lassie is an animal.

We could instead have run [Animal ?x], meaning "who's an animal?" It would then have run the first Animal rule, as before, but now it's running [Dog ?x] rather than [Dog lassie]. Then it will again start with the first Dog rule, which runs [Labrador ?x]. Where the Labrador call failed in the Lassie example, it succeeds here because its parameter is a variable. So the call to Labrador actually succeeds, and binds ?x to Bruce (i.e. it sets ?x to be bruce now). Since Labrador succeeded, the Dog rule succeeds, which means the Animal rule succeeds, but in this case, it returns back the answer that ?x=bruce, i.e. "Bruce is an animal."

Note that it gave us the answer Bruce simply because that's the first answer it ran into. If we were to swap the order of the rules for Dog, we would have looked for Collies before Labradors instead of the other way around, and so we would get the answer Lassie instead. Or alternatively, we could have added [randomly] to our rules.