SEVEN: TUPLES AND HIGHER-ORDER TASKS

To recap, we've discussed how Step executes calls to tasks – both predicates and tasks that print things – by *matching* the *parameters* specified in the call to the parameters for the different *methods* of the task. It tries a method when it can match each parameter in the call to its respective parameter in the method.

Thus far, all the values that have been passed along for parameters have been single pieces of data – a string, a variable, maybe a number. But there are times when we want to pass something more complicated as the value of a parameter. In effect, we want to package several pieces of data together as a single piece of data called a *tuple*.

This is a capability that's simultaneously absolutely crucial to any software, while also being somewhat hard to motivate to non-programmers in the abstract. So we're going to give a brief motivating example, tell you what tuples are and how to use them, and leave it at that. As you write Step code it will become clear why you need tuples.

FOR PROGRAMMERS

If you've programmed before, this is introducing a mostly familiar concept: data structures, in particular record structures (e.g. structs in C/C++). So in that sense, you're already familiar with the basic idea: you want to be able to package several data objects together as one. For example, you might want a `Person` data type that has fields of their `FirstName` and `LastName`. In *Step*, these are called *tuples*.

Logic programming languages have the same ability to represent record structures as other languages, but that functionality gets exposed differently:

- You don't need type declarations
- You create tuples just by typing them in a form that looks a little like a constructor in C++ or Java: [person "Jane" "Godall"]
- You access data in a tuple using pattern-matching

REPRESENTING BELIEFS

For example, suppose we're writing a story generator that, among other things, generates text to describe characters having a falling out. It would probably want to be parameterized by what they were arguing about. We would first want some way of finding something for them to argue about. So let's assume that we have a predicate, `Believes`, that lets us tell whether a character believes something or not. For example:

```
[predicate] [randomly]
Believes john god_exists.
Believes john money_is_important.
Believes richard god_exists.
Believes richard helping_humanity_is_important.
```

Then we could write a predicate like this (note we have to break the rules into separate lines to make it fit in word):

```
[predicate] [randomly]
Disagreement ?c1 ?c2 ?disagreement:
   [Believes ?c1 ?disagreement]
   [Not [Believes ?c2 ?disagreement]]
[end]
Disagreement ?c1 ?c2 ?disagreement:
   [Believes ?c2 ?disagreement]
   [Not [Believes ?c1 ?disagreement]]
[end]
```

Then we could have a task like:

```
FallingOut ?c1 ?c2:
[Disagreement ?c1 ?c2 ?dis]
?c1 and ?c2 had a falling out over ?dis.
[end]
```

Which might generate some text like:

```
John and Richard had a falling out over helping humanity is important.
```

That gets the idea across, but it's not awesome grammar. We can improve it by telling the system how to print things like `helping_humanity_is_important`:[1]

```
Mention helping_humanity_is_important: whether helping humanity is important.
Mention money_is_important: whether money is important.
Mention status_is_important: whether status is important.
```

This at least generates more fluent text (emphasis added):

```
John and Richard had a falling out over whether helping humanity is important.
```

A NEW KIND OF DATA: THE TUPLE

The thing is, the `Mention` rules above are almost the same. Rather than treating all these separately, we can unify them all by having a slightly more complex representation of beliefs: when we want to talk about the belief that something is important, we'll write that in the code as `[important` *something*`]`, where *something* is the thing that's supposed to be important. Then we can write John and Richard's beliefs as:

```
[predicate] [randomly]
Believes john god_exists.
Believes john [important money].
Believes richard god_exists.
Believes richard [important helping_people].
```

By bracketing together the word `important` with the thing that's important, we tell the system that they come as a unit, and they're one parameter. Now we can collapse the different `Mention` rules down to just:

```
Mention [important ?thing]: whether ?thing is important.
```

These bracketed values are called *tuples* and they let us represent complex data relatively compactly. The pattern matching process treats tuples largely the way it treats other data objects: if you match a tuple against a variable with no value, the variable is given the tuple as its value. If you match a tuple against something that isn't a tuple, that just fails, since they're different. The only thing that's slightly complicated is that when you match tuples with one another; then they must have the same number of elements and the individual elements need to match. So

[1] See "6- Adapting text to context" for a discussion of how printing works, and the `Mention` predicate in particular.

we can't match `[important money]` to `[important]`, but we can match `[important money]` to `[important ?thing]` by setting `?thing = money`.

TUPLES THAT LOOK LIKE CODE

You've actually run into tuples before. When we said:

```
[Not [Believes ?c2 ?disagreement]]
```

We were actually calling `Not` with one argument that was a tuple: `[Believes ?c2 ?disagreement]`. `Not` takes that tuple and tries to run the task it specifies, i.e. calling `Believes` with `?c2` and `?disagreement` as parameters. If the call to `Believes` succeeds, then `Not` fails; if `Believes` fails, then `Not` succeeds. `Not` just calls the task specified by the tuple, checks whether it failed or succeeded, and then does the opposite.

We call `Not` a *higher-order predicate* because it takes code as an argument rather than plain old data.[2] Higher-order tasks are very important. In the case of Not, it's a higher-order task that's built in. But you can make your own higher-order tasks/predicates.

BELIEF AND DELUSION

For example, suppose we have a predicate, `Friends`, that expresses when two characters are friends. So we would assert than John and Richard are friends with:

```
Friends john richard.
```

And we can do the usual things with `Friends` that we do with other predicates: we can ask if john and richard are friends by running:

```
[Friends john richard]
```

and we can also who John's friends are by running

```
 [Friends john ?who]
```

But now we can also represent John's *beliefs about* his friends by combining `Believes` and `Friends`:

```
Believes john [Friends john richard].
Believes john [Friends john elon_musk].
```

If we assume that in this story world, John has never actually met Elon Musk, then John has one true belief about his friends, and one false one.

We can now write a variant of `Believes`, called `Delusion`, that tells us about things a character believes that aren't true within the story world:

[2] This is a little complicated, since as we just said, the "code" we passed into `Not` was really just a tuple and tuples just are data. This is a fundamental and crucial property of computation: that code is just another kind of data. So really, what we should have said is that a higher-order predicate takes arguments that are code rather than some other kind of data.

```
Delusion ?who ?fact: [Believes ?who ?fact] [Not ?fact]
```

Now if we add the rules:

```
Disagreement ?c1 ?c2 [delusion ?c1 ?fact]:
   [Delusion ?c1 ?fact]
   [Not [Believes ?c2 ?fact]]
[end]
Mention [delusion ?who ?fact]: ?who’s crazy belief that ?fact.
Mention [Friends ?a ?b]: ?a and ?b are friends.
```

The system can generate output like:

```
John and Richard had a falling out over John’s crazy belief that John and Elon
Musk are friends.
```