TELL TUTORIAL

Ian Horswill, last updated 3/18/23

OVERVIEW

TELL is a simplified[1] logic programming language embedded in C#, meaning that TELL code literally is C# code.

Logic programming is a more **declarative** language than C#. In C#, you tell the computer what to do. In a declarative language, you tell it what an answer would look like, and leave it to the language to decide how best to achieve that. In practice, programmers do have to be mindful of how the language will try to solve the problem. But for many purposes, it's still more convenient than writing explicit code in C#.

To use TELL, you just add TELL.DLL to your project and you can mix and match TELL code with C#. While it was developed with Unity games in mind, it has no Unity dependencies. TELL is free software distributed under the MIT License, available at https://github.com/ianhorswill/TELL.

If you're already familiar with logic programming, you can safely skip to How to write TELL code in C#.

TABLE OF CONTENTS

[1] For Prolog programmers: TELL doesn't support cut or function symbols. Both of these complicate the implementation and make interoperation with native C# code more difficult. This limits the ability to do meta-programming in TELL, but it also makes it more accessible for beginners.

INTRODUCTION

Logic programming languages define their code in terms of **predicates**[2] and **rules**, rather than in terms of functions/methods. Like functions, predicates take arguments. But they express some relationship between the arguments, or if it takes only one argument, some set of things that have some property:

Concept to express	Type of predicate	Example
Membership in a set	Single-argument predicate	Integer[x], meaning "x is an integer"
Relationship between two things	Two-argument predicate	Sibling[x,y], "meaning x and y are siblings"
Function of one variable	Two-argument predicate	F[x,y] meaning "y=f(x)"
Relationship between three things	Three-argument predicate	Parents[c,m,f] meaning "m is c's mother and f is c's father"
Two-argument function	Three-argument predicate	F[x,y,z] meaning "z=f(x,y)"

For example, we might define a relationship, Sibling, that's true when two people are siblings. For example, `Sibling["Jayden", "Sora"]` might be true, but `Sibling["Sora", "Keisha"]` might be false.

QUERIES

A query is a series of predicate calls. The predicates can be passed specific values as arguments, such as `Sibling["Jayden", "Sora"]`. If all the predicates have specific values for their arguments, then we're just asking if all the predicates are true of their arguments.

However, we can also pass "logic variables" as arguments to predicates, and these behave somewhat differently than normal programming language variables (see Logic variables and pattern matching). If you run a query that contains variables, you are asking the system if there's a set of values for the variables that **will make all the calls true**. For example, if we assume the Parent predicate is true when the second argument is a parent of the first, then we might ask if Keisha is Jayden's aunt by asking:

```
Parent["Jayden", x], Sibling[x, "Keisha"]
```

meaning: "can you find an x who is Jayden's parent and Keisha's sibling?" If so, the query is true, otherwise false.

Much of the usefulness of logic programming comes from the fact that it can report back to you the values of the variables it found. Moreover, if you don't like the value that it found, you can ask it to try to find a different value, or to find all the possible values. So, a single predicate can be used in many different ways:

Query	Meaning
`Parent["Sora", x]`	Who is a parent of Sora?
`Parent[x, "Sora"]`	Who is a child of Sora?
`Parent["Jayden", "Sora"]`	Is Jayden Sora's child?
`Parent[x, y]`	Give me a parent/child pair (I don't care who)
`Parent[x, y], Parent[y, z]`	Give me a child/parent/grandparent triple (I don't care who)

[2] Predicates are essentially the same as relations, although they're represented differently than they are in databases.

`Parent[x, y], Parent[y, "Sora"]`	Give me Sora's kid/grandkid

Queries are the core of the code that gets run. When your C# code calls into TELL, it's running a query, either to test the truth of a proposition, or to solve for the values of one or more variables. However, queries are also the basis of rules, which are the core of most logic programming code.

RULES

A TELL rule says "this is true if those things are all true." In particular, it says this "this predicate with these arguments is true if this query is true." While it's possible to define predicates directly in terms of C# code, predicates are typically defined in terms of rules; this is where logic programming gets its power. A predicate is true for some set of arguments if one of its rules says it's true. Otherwise, it's false.

For example, if we've defined the `Parent` and `Sister` predicates, we can define an Aunt predicate by the rule:

```
Aunt[c, a].If(Parent[c, p], Sister[p, a]);
```

This says "a is c's aunt **if** p is c's parent **and** a is p's sister," i.e. "an aunt is a child's parent's sister." Rules are assumed to apply for any values of the variables that appear before the `If()`. Given that, if we want to know who Jayden's aunt is, we can substitute Jayden in for c in the rule above, and run the resulting query:

```
Parent["Jayden", p], Sister[p, a]
```

That's the basic strategy of classical, "Prolog-style" logic programming systems: if you want to know if some predicate call is true, or equivalently, to find values of the variable in it, substitute its arguments into each of the predicate's rules, and run their `If()` queries. If one of the rules works, we're done (the call "succeeds"). If not, the call is false (aka it "fails").

Suppose we don't have a `Sister` predicate defined, but we have `Sibling` and `Female`. Then we can define the former in terms of the latter:

```
Sister[x,y].If(Sibling[x,y], Female[y]);
```

Now, if we run the query:

```
Aunt["Jayden", a]
```

The system essentially substitutes "Jayden" into the body to get:

```
Parent["Jayden", p], Sister[p, a]
```

And substitutes p for x and a for y in the Sister rule to get:

```
Parent["Jayden", p], Sibling[p, a], Female[a]
```

And then finds values for p and a that make that work.

Now, let's look at a more complicated predicate. Suppose we want to define ancestry in terms of the `Parent` predicate. We can say:

```
Ancestor[x,y].If(Parent[x,z], Ancestor[z,y]);
```

This says "y is x's ancestor **if** z is their parent **and** y is z's ancestor". Again, when a rule uses a variable, it is implicitly saying the rule applies for all possible values of its variables. So this works, no matter what x and y we choose, and for any z, so long as z is actually x's parent and a descendant of y.

This is a recursive rule. And so we need another rule to be a base case. So we usually say that a person is their own ancestor:

```
Ancestor[x,x].If();
Ancestor[x,y].If(Parent[x,z], Ancestor[z,y]);
```

Note that this new rule has no query in the `If()` part; it's true unconditionally. Rules like this are often called "facts" in logic programming, and so in TELL, we would usually replace the `If` with `Fact` just to make this clearer:

```
Ancestor[x,x].Fact();
Ancestor[x,y].If(Parent[x,z], Ancestor[z,y]);
```

This says "any x is it's own ancestor."

Of course, you could have written this as a normal recursive function to test whether y was x's ancestor:

```
bool IsAncestor(Person x, Person y) => x == y || IsAncestor(Parent(x), y);
```

But our rules don't just let us test whether two specific people are ancestors:

```
Ancestor["Jayden", "Sora"]
```

They also let us ask about for Jayden's ancestors:

```
Ancestor["Jayden", x]
```

Or Sora's descendants:

```
Ancestor[x, "Sora"]
```

Or all ancestor/descendant pairs:

```
Ancestor[x, y]
```

Using the same pair of rules.

PRIMITIVE PREDICATES

Not all predicates are defined in terms of rules. You can also specify C# code to run when a predicate is run. These are called primitive predicates. You can use these to interface to the native data structures of your game. We'll talk about how to write these later.

DECLARATIVE SEMANTICS: HOW YOU SHOULD PRETEND TELL CODE IS EXECUTED

Here's one way to define a logic programming language like TELL: a TELL query is "true" if there's a set of values for its variables that make it true given the rules you've provided. This definition says nothing about how the system finds those values, and in many ways that's the point of using a language like TELL. On a good day, you give it a bunch of true statements, you then ask it questions about other statements, and it's TELL's problem to worry

about how to answer them. That's what it means for a language to be **declarative**, and this definition of the meaning of a TELL program is called its declarative semantics.

DECLARATIVE VS PROCEDURAL SEMATNICS

Of course, in practice TELL uses a very specific algorithm to try to find those values. Moreover, it's an incomplete algorithm, meaning it can miss solutions. In practice, programmers working with languages like TELL and Prolog often have to design around the algorithm's its blind spots. That is, they have to keep the language's **procedural** semantics in mind while programming. The procedural semantics infect the declarative semantics, so to speak. And, in fact, you can even use your knowledge of the algorithm to trick TELL into running loops or forcing it to do things in a specific order. Sometimes you just have to do that to make it work. But even when you do, it's a really bad idea to design it so that the people calling your code have to think about how it works internally. If they do, then you should probably be writing that part of your code in C# rather than TELL.

PROCEDURAL SEMANTICS: HOW TELL CODE IS ACTUALLY EXECUTED

Okay, so what's the real algorithm? Here's the simple version:

- Predicates are like procedures
- Rules are like methods for a procedure
- A query to test the truth of a predicate with arguments is a call to that predicate/procedure
- When a predicate is called, the system tries each rule (method) for the predicate, in the order they were declared, until one works
- When it tries a rule, it recursively tries to prove each of the calls in the If() part, from left to right
- As it goes through this process, it accumulates values for the different variables.

GOALS, RULES, AND SUBGOALS

From here on it, will be useful to refer to predicate calls as **goals**, because that's ultimately what they are: they're things the system is attempting to prove true. It tries to prove them by **matching** them to rules. In the rule:

```
Ancestor[x,y].If(Parent[x,z], Ancestor[z,y]);
```

the left-hand side of it, `Ancestor[x,y]`, is called the **head** of the rule. And the right-hand side, `Parent[x,z], Ancestor[z,y]`, is called the **body**. The body calls are **subgoals** to prove when we try to use this rule to prove the head. When we call `Ancestor["Jayden", w]`, it matches the goal, `Ancestor["Jayden", w]`, with the head, `Ancestor[x,y]`, to determine that the rule's x variable must be "Jayden", and the rule's y variable must be the same as the goal's w variable:

x="Jayden"
z=w

It can then substitute that into the rule to get a series of subgoals to try:

```
Parent["Jayden",z], Ancestor[z,w]
```

So now it calls `Parent["Jayden",z]`, and suppose Jayden's mother is Sora. Then we get back z=Sora, which means our subgoals changed into:

```
Parent["Jayden","Sora"], Ancestor["Sora",w]
```

and we've already established that the first of these is true. So then the question is who is an ancestor of Sora? We have the rule:

```
Ancestor[x,x].Fact();
```

whose head is `Ancestor[x,x]`, and which has no subgoals, meaning it's true unconditionally. The system matches the subgoal `Ancestor["Sora",w]` against the head `Ancestor[x,x]`. To make that work, the first arguments have to be the same between the two, so:

x="Sora"

but the second arguments have to be the same as well:

x=w

But these together mean that w="Sora", which answers our original `Ancestor["Jayden", w]`, goal; we know w="Sora", i.e. "Sora is one of Jayden's ancestors," which is correct.

That's a basic overview of how it works. In a sense, it's very similar to the normal call-and-return structure of a normal programming language. But there are a couple of things that are very different that are worth discussing in a little more detail.

LOGIC VARIABLES AND PATTERN MATCHING

TELL's variables behave differently than C# variables. C# variables are ultimately just a place in memory to store a value, and that place always has a value one way or another. You might be able to set it to `null`, but that's just another value.

TELL variables, known in programming languages as **logic variables**, start out with unknown values. Then, as we saw, the process of matching goals and heads of rules causes the system to learn that particular variables must have particular values, or that pairs of variables must have the same values:

x="Jayden"
z=w

These are called **variable bindings**. TELL's variables always begin in an "unbound" state, meaning they haven't yet been given a value. Then, during the execution process, they generally acquire values through bindings found by matching. Once a variable is bound to a specific value, any further matching uses that value; it can't "reset" the value of the variable.

So conceptually, logic variables are "write once." And when you're reasoning about your code, that should be how you think about it. Under the hood, it's a little more complicated; bindings do get abandoned during execution, so let's talk about that now.

NONDETERMINISM

There are a number of points where TELL has to make choices. When you call a predicate, it must guess what rule to use for the predicate. But that rule may call other predicates, and it has to guess what rule to use for each of those. So executing a query typically results in a **tree** of possible choices.

TELL is a "nondeterministic" language, meaning that you can pretend it **always guesses correctly**: it always chooses a path through the choice tree that "works," provided there is one.

In reality, it's trying things and giving up when they **fail**, and then trying something different; it searches the tree of possible choices.[3] But when it gives up on an attempt, it removes all traces of the attempt, other than to remember that path failed. In particular, when it gives up on a rule, it throws away any variable bindings accrued during the execution of the rule. The variables are returned to their states before the rule.

In the end, the system looks as if it had made all the right guesses all along; only the bindings from the successful execution path remain.

CALLING, FAILING, AND RETRYING

We've just slipped two concepts in the backdoor here that don't typically appear in normal languages: **failure** and **retrying**. What happens if a goal can't match any rules, or we've gone through all the rules already? That goal **fails**: it forces the system to choose another path through the tree of choices. Let's look at an example:

```
IsA[“Fred”, “cat”].Fact();
IsA[“Gerry”, “cat”].Fact();
IsA[“Sally”, “cat”].Fact();
IsA[“Fido”, “dog”].Fact();
IsA[“Joan”, “dog”].Fact();

Color[“Fred”, “grey”].Fact();
Color[“Fido”, “brown”].Fact();
Color[“Gerry”, “brown”].Fact();
```

Remember that `.Fact()` means a rule with no subgoals. Suppose we want to find a brown cat:

```
IsA[x, “cat”], Color[x, “brown”];
```

The solution to this is x="Gerry", and you demonstrate that using the two rules in yellow. To produce the illusion that it guessed the right rules all along, TELL, tries the rules one at a time. TELL starts with no variable bindings, and tries the first goal, `IsA[x, “cat”]`. It matches the goal against the first IsA rule. They match, yielding the binding:

x="Fred"

Now it tries, the second goal, `Color[x, “brown”]`, except x="Fred" so the goal is really `Color[“Fred”, “brown”]`. It tries to match that against all the Color rules, but none of them match; we say the goal `Color[“Fred”, “brown”]` **fails**.

[3] It uses a depth-first search, in case you were wondering.

Now the system needs to find a different solution to the first goal, `IsA[x, "cat"]` because its first solution didn't work for the second goal. When a goal fails, it "**retries**" the goal before. IsA **undoes** the binding x="Fred" that it found from the first rule, and picks up with the next rule. That matches with the binding:

x="Gerry"

Now the IsA call has succeed for a **second time**. We then try the `Color[x, "brown"]` goal again, but now it's really `Color["Gerry", "brown"]`. Color tries to match that goal against its rules. It doesn't match the first two, but it matches the last one. So now the second goal has also succeeded, and we have a solution: x="Gerry".

BACKTRACKING CONTROL STRUCTURE

So now you have the basic execution algorithm. It moves through rules in order, trying each one. Within a rule, it tries each subgoal, in order. As it tries things, it accumulates bindings, causing variables to go from unbound to bound. Once bound, variables don't change their value unless some kind of failure retracts the binding.

This structure of moving forward when you succeed and backward when you fail, with one subgoal potentially succeeding many times, is called **backtracking**. In particular, the process of backing up to a previous goal, and throwing away some bindings is called backtracking.

PREDICATES AS ITERATORS

Note: if you don't know about coroutines and/or you find this confusing just skip over it.

Backtrackable calls behave like coroutine iterators in C# (the stuff where you return `IEnumerable` using `yield return`). Like an iterator, a goal can generate multiple return values, but can also say "sorry, I don't have anything." An easy way to implement a language like TELL would be to have goals effectively be iterators that take in a set of bindings and return a coroutine that generates new sets of bindings, one for each new solution. Then we could write a query as something like this:

```
Bindings FindBrownCat(Bindings originalBindings) {
   foreach (var newBindings in IsA[x, "cat"].Solve(originalBindings))
      foreach (var finalBindings in Color[x, "brown"].Solve(newBindings);
         return finalBindings;
}
```

We have some datatype, `Bindings`, that somehow represents a set of variable bindings. `FindBrownCat` takes a set of bindings currently in effect and gives us a new set of bindings that includes a binding for x. Or if we want to find all the cats, we can do:

```
IEnumerable<Bindings> FindBrownCats(Bindings originalBindings) {
   foreach (var newBindings in IsA[x, "cat"].Solve(originalBindings))
      foreach (var finalBindings in Color[x, "brown"].Solve(newBindings);
         yield return finalBindings;
}
```

For a number of reasons, primarily debuggability, TELL doesn't use this implementation strategy (although UnityProlog and YieldProlog use variants of it). But it's nice because it explains backtracking in terms of more conventional control structures available in languages like C# and python.

HOW TO WRITE TELL CODE IN C#

TELL is an "embedded" language. TELL code is written in the form of C# code that happens to call into the TELL library to make predicates, add rules to them, and call them. We've taken pains to let TELL code look as natural as possible inside other C# code, but we're limited by what kinds of overloading C# allows. For example, predicates are called using square brackets rather than parentheses because C# lets you overload the former but not the latter.

ADDING TELL TO YOUR GAME

To use TELL in a Unity game, either add TELL.DLL to your Assets folder, or copy the source files into it. The former will let compilation run faster, the latter will let the debugger step through the TELL interpreter if you want to.

Having added TELL to your project, just add:

```
using static TELL.Language;
```

to any .cs files that use TELL.

STRONG TYPING

Like C#, TELL predicates and variables are typed. So `Var<int>` is the type of a TELL variable whose value is an integer, while `Var<string>` the type of one that is a string. `Predicate<int>` is the type of a one-argument predicate whose argument is an integer, `Predicate<int,string>` the type of a two-argument predicate that takes an integer and a string. We've taken pains to shield you from having to explicitly declare types any more than necessary.

MAKING VARIABLES

The main exception is when you create TELL variables. You generally can't ignore explicitly specifying a type for those. But all you have to specify is a type and a name:

```
new Var<type>("name")
```

The name is there so that if you get an exception at run-time you can poke around in the stack and see that it's the float variable x and not some other float variable.

Of course, it's a little confusing to use new to make a variable. But TELL's variables can't be C# variables; they need to be C# objects so that you can use them to build rules at run-time that get stored inside the interpreter. Nevertheless, you still need to store one in a C# variable so you can use it in your C# code, so what you really want to say to declare a variable is:

```
var name = new Var<type>("name");
```

This is rather cumbersome, but you can abbreviate it to:

```
var name = (Var<type>)"name";
```

which saves some typing. So your TELL program might begin with a bunch of declarations such as:

```
var person = (Var<Person>)"person";
var name = (Var<string>)"name";
var age = (Var<int>)"age";
var location = (Var<Vector3>)"location";
```

and so on. The bad news is that this is the least cumbersome way we were able to make variable declarations work. The good news is that once you make one of these variables, you can use it in as many rules and queries as you want; variables just represent local names within a rule or goal, so they don't get confused between rules.

MAKING PREDICATES

Predicates are also C# objects. So after you create one, you'll also want to store it in a variable. As we said, they're strongly typed, so they know what type to expect for each argument. Fortunately, you can usually avoid explicitly specifying the type of a predicate when you create it. To create a predicate, just say:

`var` *name* `= Predicate("`*name*`",` *argument-vars* `…);`

You provide TELL variables to represent its arguments. Since these are TELL variables you already created, this lets the system infer the type of the predicate from the types of the arguments. So if, for example, we wanted to define a predicate that was true when a particular NPC had a particular name, we might say:

```
var Name = Predicate("Name", person, name);
```

Since it already knows that `person` is a variable of type `Person` and `name` is of type `string`, it then knows that the predicate `Name` is of type `Predicate<Person,string>`, i.e. a predicate that takes a `Person` and a `string` as arguments. If we defined appropriate rules for this predicate, then we can do things like call it with a variable for the second argument to ask it to solve for the name of a particular person, or to call it with a variable for the first argument to solve for what person has a given name.

CALLING PREDICATES

You make a goal by applying the [] operator to a predicate:

- `Name[person, "Fred"]`
- `Age[person, age]`
- `Age[person, 10]`

Making a goal doesn't call the goal. It makes a data structure representing the call. You can place it in a rule, in which case it will be called if/when the rule is used, or you can call it immediately from C# by calling one of several different methods on goals:

- *goal* (e.g. `if (`*goal*`) …`)
  Runs goal and returns true if it succeeds, false if it fails. This is just an implicit type conversion rule from `Goal` to `bool` that runs `IsTrue`.
- *goal*`.IsTrue()`
  Run it and return true if it succeeds, false if it fails.
- *goal*`.IsFalse()`
  Same, but with the return value reversed.

- *goal*.SolveFor(*variable*)
  Run it, and return the value of *variable* from the resulting solution. *Variable* must appear in the goal itself. If *variable* is unbound in the solution or the goal fails, this will throw an exception.
- *goal*.SolveForAll(*variable*)
  Same, but it finds all solutions and returns a list of the values for each solution.
- *goal*.Solutions
  Returns a list of tuples of the values of the arguments from *goal* in each solution. If *goal* has one argument, it will return a list of that argument. If it has two arguments, it will be a list of tuples with the values of each argument in each solution. And so on.

WHAT IS A TERM<T>?

Note: you can safely skip this, but since you're likely to see the type Term<*T*> appear in popups in your code editor, we'll explain it here in case you're wondering.

In logic, the arguments to predicates are called "terms" and they're either constants, variables, things called function expressions, which TELL doesn't support (use Prolog for that).

TELL is ultimately an interpreter and so it stores what amounts to a parse tree for your program. When you call a predicate using the [] operator, it creates an object of type Goal, which is effectively the parse tree of the call. That's why TELL variables have to be objects in C# - they're the things that literally get stored in the parse tree.

Say we have a predicate, P, which is of type Predicate<int>, meaning it takes one argument and that one argument should be an int. That means we want to be able to pass it an actual int, or a TELL variable that's an int TELL variable. C# doesn't let you declare the type of an argument to be "int or Var<int>". So we deal with this by making the actual argument type for the predicate be Term<int>, which has two subclasses, Var<int> and Constant<int>. You've never had a reason to see the Constant<T> type here because there's an implicit type conversion rule that says that if you pass a *T* into something that wants a Term<*T*>, then the compiler can quietly replace the *T* value with new Constant<*T*>(*the original value*). A Constant<*T*> object is just a wrapper for the original object to let the interpreter see in the parse tree that it's a constant rather than a variable and what that constant value is.

So for most purposes, you can ignore the distinction between Term<*T*>, Constant<*T*>, and just *T*. But if you see the Term or Term<*T*> type show up in documentation, now you know why it's there rather than just *T*.

ADDING RULES

As we said before, you can add a rule to a predicate by saying:[4]

predicate[*args*…].If(*subgoals* …);

That's all you need to know. However, there are a couple of quality-of-life features that can aid readability:

- *predicate*[*args*…].Fact();
  Equivalent to an If() with no subgoals. This just looks a little less weird than *predicate*[*args*…].If();

[4] Technically, this is calling *predicate* to get a Goal, then calling the Goal's If() method, which tells the goal to make a new, internal Rule object that has the goal as its head and the subgoals as its body, then add that Rule object to the original *predicate*.

- `Predicate("name", argVariables…).If(subgoals);`
  You can combine the creation of the predicate and the addition of the rule into one expression if the head of the rule was going to use the *argVariables* anyway.

READING RULES FROM A CSV FILE

You can also load facts (rules without subgoals) from a CSV file using the `LoadCSV(string path)` method of the predicate, e.g.:

```
var People = Predicate("People", name, age).LoadCSV("people.csv");
```

The CSV file should have as many columns as the predicate takes arguments. It should also have a header row with column names that match the names of the variables passed into the constructor for predicate (case is ignored). So in the example above, it would need to have a header row that read: Name,Age or name,age.

The cells in the CSV will be converted from string form based on the declared types of their respective arguments in the predicate. However, if the cell is the fixed string "_", then it will be read as a wildcard (a variable) and provide a rule that matches for any value of that argument.

The built-in system parses the types: `string`, `int`, `uint`, `double`, `float`, `bool`, and `enumerated types`. If these aren't enough, you can define custom converters for your own types using:

```
TELL.Interpreter.CsvReader.DeclareParser(Type t, Func<string, object> parser);
```

This needs to be run before it will take effect in a call to `LoadCSV()`.

PASSING DATA FROM YOUR GAME INTO TELL

The point of embedding TELL in C# is to make it easy for it to interoperate cleanly with native game code written in C#. You'll do that in two main ways: by passing game data into the predicate calls, and by making new **primitive predicates** that call directly into your game code.

PASSING GAME DATA TO PREDICATES

The arguments to a predicate in a goal can be any C# expression you like, so long as it's of the right type. So you can pass in data from your game however you see fit. If it's a TELL variable (type `Var<T>`), it will treat it as a variable it needs to find a value for. If it's any other value, it treats it as a constant: that specific data object is the argument to the predicate. Suppose we have our TELL variables from above, which I'll put in boldface:

```
var person = (Var<Person>)"person";
var age = (Var<int>)"age";
```

And suppose we also have a plain, old C# variable, which I'll put in italics:

```
Person player = Person.Player;  // Assumes this returns a Person object
string str = "John Doe";
```

Then this is what each of these calls mean:

- `Age[person, 29].SolveFor(person)`
  Find me a character who is 29 years old.
- `Age[player, 29]`
  Is the `Person` object stored in the player variable 29 years old?
- ~~`Age[player, 10].SolveFor(player)`~~
  Won't compile: *player* is of type `Person`, not `Var<Person>`.
- `Age[player, age].SolveFor(age)`
  Get me the age of the *player* character.

So there's a difference between passing in the value of a normal C# variable and passing in one that happens to hold a TELL variable. In the former case, the variable is holding a `Person` object and so that gets treated by the call as a constant. In the latter case, it holds a `Var<Person>` and so gets treated as something that can be bound in pattern matching and solved for.

Important: if you include `Age[player, 29]` in a rule, then you are hard-coding the value of *player* at the time the rule is defined. You are *not* asking it to look up the value of *player* anew each time the rule is called. To write something that looks up a value every time it's called, use a primitive predicate. To learn to do that, read on.

DEFINING PRIMTIVE PREDICATES

A primitive predicate is **defined directly in C#** rather than through rules. You provide a delegate (C# method) and each time the predicate is called, the predicate calls the delegate and the delegate defines what to do. The true delegate type used inside the interpreter is called `TELL.Interpreter.Prover.PredicateImplementation` and requires the programmer to understanding structure of the interpreter. So we don't recommend using it. We've provided a set of wrappers you can use to handle the common cases that most people want to use in their games.

Important: TELL predicates defined by rules can be called with an input that is an unbound variable (i.e. that hasn't yet been determined). However, C# functions don't generally understand this. So primitive predicates generally **throw exceptions** if you call them with unbound variables for the C# function's inputs.

SIMPLE TESTS

If you want to expose a Boolean test from your code to TELL, use the `SimpleTest` type:

```
var name = Predicate("name", (SimpleTest<T1, …, Tn>)((args…) -> Boolean));
```

This makes a predicate that takes the specified number of arguments of the specified types, and calls the specified delegate with the values passed in the call. It succeeds if the delegate returns true, and fails if it returns false. It throws an exception if any of the inputs are unbound variables.

For example, we can define a method that tests if an integer is odd by saying:

```
var Odd = Predicate("Odd", (SimpleTest<int>) (n -> (n&1)>0));
```

Now let's say we call `Odd[x]`. The declarative semantics of this are that this will succeed exactly when x is an odd number. However, the exact runtime behavior as we've defined it depends on the binding of the variable x:

x bound to	Behavior
1 or other odd number	**Call succeeds**
2 or other even number	Fall fails
Nothing (unbound variable)	**Exception: there's no value to call the delegate with**

If Odd were a classical logic programming predicate, then calling it with an unbound variable would bind it to an odd number, and if we kept retrying it, it would eventually generate every possible odd number. Since there are a lot of odd numbers, this is almost certainly not the behavior we want. So the wrappers don't attempt to support this. However, if you want to write predicates that can do this kind of thing, see Mode dispatch,below, for a discussion of how to write predicates that can handle both bound and unbound arguments.

SIMPLE FUNCTIONS

If you have a conventional C# function that takes inputs and returns a value, you can wrap it in a TELL predicate that takes the inputs as its first arguments and returns the output as its last:

```
var name = Predicate("name", (args…) -> value);
```

For example, if the native `Person` class in our game has a `Name` field, we could write an accessor for it as a TELL predicate by saying:

```
var Name = Predicate("Name", (Person p) -> p.Name);
```

Now say we call `Name[person, name]`. We should think of this as having the semantics that it succeeds exactly when person is a person and name is their name. However, the exact runtime behavior depends on the binding states of its inputs:

Person bound to	Name bound to	Adds binding for name?	Succeeds or fails?
Person object	**Unbound**	**Yes; to person's name**	**Succeed**
Person object	**That Person's name**	**No**	**Succeed**
Person object	Any other string	No	Fail
Unbound	**Doesn't matter**	**No**	**Exception**

Again, this blows up if its first argument is unbound because then there's nothing to pass to the delegate. See Mode dispatch for a discussion of how to write predicates that can handle both bound and unbound arguments.

ENUMERATORS

Now suppose that a Person, in addition to their name, has a list of aliases. We could write that using the technique above:

```
var Aliases = Predicate("Aliases", (Person p) -> p.Aliases);
```

Now `Aliases` is a predicate whose first argument is a person and its second argument is a list of aliases. That forces us to work with the list as an object, and that's usually inconvenient in logic programming.

It's generally more useful to have a predicate called `Alias` (singular) whose second argument is a single string (a single alias), and that is true if that specified person has that specified alias. We can do that by saying:

```
var Alias = Predicate("Alias",
                      (Enumerator<Person,String>)((Person p) -> p.Aliases));
```

Now if we call `Alias[person, name]`, it should work whenever name is one of person's aliases. It's exact runtime behavior is:

Person bound to	Name bound to	Adds binding for name?	Succeeds or fails?
Person object	**Unbound**	**Yes; to person's first alias. Retries generate successive aliases**	**Succeed**
Person object	**One of the person's aliases**	**No**	**Succeed**
Person object	A string that isn't one of their aliases	No	Fail
Unbound	**Doesn't matter**	**No**	**Exception**

If we call it with a person and an unbound variable, it will bind the variable to the first alias, and continue. If we later fail and retry the call to `Alias`, we get the next alias. If need be, it will succeed once for each possible alias.

Again, this throws an exception if the first argument is unbound. To write something that works with all possible inputs, see Mode dispatch.

MODE DISPATCH

C# functions are written to expect a particular pattern of inputs and generate a single output. To adapt this to the more general system supported by logic programming systems, we need to dynamically select different C# functions based on what arguments are bound (inputs) and what arguments are unbound (outputs). As we said, you can provide an implementation delegate for a primitive predicate that can access the interpreter data structures directly and do whatever it wants. But that's a pain at the best of times and it also requires learning the internals of the interpreter.

To keep you from having to learn the internals of the interpreter, we've provided you with a function called `ModeDispatch` that takes separate C# functions for each possible configuration of bound and unbound for each input and returns an implementation delegate that will choose the right one to run at runtime. The problem with this is that for n inputs, there are 2^n different cases. So we currently only support 1- and 2-argument predicates. This is mostly just because I don't expect anyone to actually want to write all 8 necessary C# functions for a 3-argument predicate. But if someone really wants that, it's easy enough for me to write the 3-argument or even the 4-argument case.

DEFINING A COMPLETELY GENERAL 1-ARGUMENT PREDICATE

To write a 1-argument predicate that that can both test (take a bound argument) and generate (take an unbound argument) you need to specify both a function for testing and a function for generating:

```
var name = Predicate<T>("name",
                        ModeDispatch<T>(Func<T, bool> tester,
                                        Func<IEnumerable<T>> generator);
```

You fill in *name*, *T*, *tester*, and *generator*. In most cases, the compiler will be smart enough to infer *T*, but I'm including all the type annotations here to be clear.

If the resulting predicate is called with a value (i.e. a constant or a bound variable), then it will call *tester* with that value. Tester takes a *T* and returns a bool. The predicate will succeed if the bool is true, and fail otherwise.

If the predicate is called with an unbound variable, then it runs *generator* instead. Generator takes no inputs and returns an IEnumerable<T>. The predicate will bind its argument to the first element of the enumerable and

succeed (continue). If it's forced to retry, it will move on to the next element and succeed again, until it runs out of elements in the enumerable.

Here's a common example. You have some built-in data type and you want to define it as a predicate. For example, suppose you're writing a Unity game, and the game has a component called NPC. You can write a predicate called Character (I won't call it NPC to avoid name collisions) like this:

```
var Character = Predicate<Component>("character",
          ModeDispatch<Component>(c => c is NPC,
                                  () => UnityObject.GetObjectsOfType<NPC>());
```

If you call this with a component, it tests whether it's an NPC. If you call it with an unbound variable, it sets it to an NPC and lets you enumerate all NPCs if you backtrack. Note we could also have written this as taking NPC as its argument type, but then it's not possible to call it with a non-character, obviating the need to handle both testing and generating.

DEFINING A COMPLETELY GENERAL 2-ARGUMENT PREDICATE

Two argument predicates work the same way, we just need to specify four different C# functions rather than 2:

```
var name = Predicate<T1,T2>("name",
                        ModeDispatch<T1,T2>(Func<T1, T2, bool> tester,
                                            Func<T1, IEnumerable<T2>> generator2,
                                            Func<T2, IEnumerable<T1>> generator1,
                                            Func<IEnumerable<(T1, T2)>> generator);
```

The first gets called when both arguments are specified (bound), the second when only the first is specified (and it generates possible values for the second), the third when only the second is specified (and it generates possible values for the first), and the last is for when no inputs are specified, and it generates all possible pairs of values for the two arguments. You can specify `null` for any of these, in which case the predicate will throw an exception should that unhandled binding pattern come up at runtime.

Here's how we would write a completely general version of the `Alias` predicate:

```
Var Alias = Predicate<Person,string>("Alias",
             ModeDispatch<Person,string>(
               // Does this person have this alias?
               (person, name) => person.Aliases.Contains(name),
               // What are the aliases of this person?
               person => person.Aliases,
               // What persons have this alias?
               name => Person.AllPersons.Where(p => p.Aliases.Contains(name)),
               // Give me every (person,alias) pair.
               () => Person.AllPersons.SelectMany(
                                       p => p.Aliases.Select(a => (p, a)))));
```

BUILT-IN PREDICATES

The following primitive predicates are built into the system. They're defined in the static class `TELL.Language`.

Note:

- The type `Term<T>` can effectively be considered to be the type T.
- Generic predicates are called with () rather than [] because of idiosyncrasies of C#.

COMPARISONS

- `Same<T>(Term<T> x, Term<T> y)`
  `x == y`
  True if the two arguments are equal. However in logic programming, this really means "true if you can match them", which means that if one of them is unbound, it will make them be the same by binding them together, and then succeed since they're the same now.
- `Different<T>(Term<T> x, Term<T> y)`
  `x != y`
  True if the two arguments are **not** equal. However in logic programming, this really means "true if you **cannot** match them", which means that if one of them is unbound, it will make them be the same by binding them together, and then fail, because they're not different anymore.

DEBUGGING

- `Break<T>(Term<T> arg)`
  Forces your game to enter the debugger. You will then be able to see where you are in execution by inspecting the C# stack, and look at the value of arg.

HIGHER-ORDER PREDICATES

Higher-order predicates are ones that take goals as arguments and run them for you.

- `Not[Goal g]`
  `!g`
  True if g is false, false if it's true. In other words, it runs g and fails if g succeeds and vice-versa.
- `[And` *goal* …`]`
  True if all goals are true. This is an unusual predicate in that it takes a variable number of arguments.
- `Once[Goal g]`
  True if g is true, but only generates the first solution to g. It will fail if you attempt to retry it.

AGGREGATE PREDICATES

- `Sum[Var<float> sumValue, Goal generator, Var<float> totalValue]`
  True if `totalValue` is the sum of value of `sumValue` across all solutions to `generator`. For example, `Sum[m, Member(m, list), total]` would set total to the sum of all the elements of `list`.

- `Min[Var<float> value, Goal generator]`
  True if `value` is the smallest of value it takes across all solutions to `generator`. For example, `Min[m, Member(m, list)]` would set `m` to the smallest element of `list`.

- `Max[Var<float> value, Goal generator]`
  True if `value` is the largest of value it takes across all solutions to `generator`. For example, `Min[m, Member(m, list)]` would set `m` to the smallest element of `list`.

- `Minimal<T>(Var<T> arg, Var<float> utility, Goal g)`
  Finds the value of `arg` that minimizes `utility`, from all solutions to the goal. In other words, it finds all solutions to the goal, remembers the values of `utility` and `arg` for the one with the lowest value of `utility`.

- `Maximal<T>(Var<T> arg, Var<float> utility, Goal g)`
  Finds the value of `arg` that maximizes `utility`, from all solutions to the goal. In other words, it finds all solutions to the goal, remembers the values of `utility` and `arg` for the one with the highest value of `utility`.

"META-LOGICAL" PREDICATES

Meta-local is a terrible name, but it's the standard one. These are predicates that let you inspect the state of the interpreter. In particular to ask if a variable is bound. You need that sometimes, for example to bullet-proof code that will loop infinitely if called with an unbound variable.

- `Unbound<T>(Term<T> arg)`
  True if arg is a variable that is still unbound at the time of the call.
- `Bound<T>(Term<T> arg)`
  True if arg is not a variable or if is a variable bound to a value at the time of the call.

OTHER

- `Member<T>(Term<T> element, Term<IList<T>> list)`
  True if element is a member of list. List must be bound, but element can either be bound, in which case Member will test membership in the list, or unbound, in which case it will generate elements of the list.

COMMON PROBLEMS WITH LOGIC PROGRAMS

The most common bugs in logic programs fall into a few categories. Many are just literally bugs in the programmer's logic: edge cases, and the like. These aren't the language's fault. But there are also bugs that are due to the difference between the declarative semantics that these languages aspire to and the procedural semantics they actually have. Here are the most common ones.

ORDER OF RULES MATTERS

TELL tries your rules one at a time, in order. That means, for example, that if you write a recursion, the rule for the base case needs to come first. The predicate we discussed earlier:

```
Ancestor[x,x].Fact();
Ancestor[x,y].If(Parent[x,z], Ancestor[z,y]);
```

works. But if we reverse the order of the rules:

```
Ancestor[x,y].If(Parent[x,z], Ancestor[z,y]);
Ancestor[x,x].Fact();
```

We can get an infinite recursion, because it will always start by trying the recursive case, which will itself start with the recursive case, and so on, never getting to the base case. This won't always generate an infinite recursion, but if you call it with both arguments unbound it will recurse infinitely because the arguments are still unbound when it gets to the recursive call. That means the recursive call is effectively solving the same problem as the original call, and the system will keep reducing the current problem to another one that's basically the same problem.

The general rule of thumb is to **put recursive rules at the end**.

ORDER OF SUBGOAL MATTERS

Within a rule, TELL always executes subgoals left-to-right. That leads to a couple of problems.

INFINITE RECURSION

Again, the definition of ancestor that works:

```
Ancestor[x,x].Fact();
Ancestor[x,y].If(Parent[x,z], Ancestor[z,y]);
```

depends on the recursive rule calling `Parent` first to get a specific value of z to consider and only then recursing. If we reverse these:

```
Ancestor[x,x].Fact();
Ancestor[x,y].If(Ancestor[z,y], Parent[x,z]);
```

We start with the recursive subgoal `Ancestor[z,y]` which, since z is unbound at that point in time, will look for any node that has y as an ancestor. That might actually work. However, suppose we started with both x and y unbound. Then we're asking "find me something that is an ancestor of something else." Good so far. But now the recursive call is with another unbound variable, z, and so we're again effectively asking "bind me something that is an ancestor of something else," meaning we've reduced a problem to itself, and will fall into an infinite recursion.

The general rule of thumb is to **put recursive calls at the end of your rules**.

CALLING PRIMITIVES WITH UNBOUND VARIABLES

Not all primitives can handle unbound arguments, and those that can usually can't handle arbitrary unbound arguments. For example, the `Odd` primitive that we defined earlier only works when you give it a specific number to test for oddness. That means, if we for some reason want to find a person whose age is an odd number:

```
Person(person), Age(person, age), Odd(age)
```

The query will work because by the time we call `Odd`, `age` has been filed in by the call to `Age`. But if we reverse the order of the last two subgoals:

```
Person[person], Odd[age], Age[person, age]
```

Then we'll get an exception because `age` doesn't have a value by the time `Odd` is called.

EXTRANEOUS SOLUTIONS

Suppose you have a rule like this:

```
FavoriteFood[person, "pizza"].Fact();
```

i.e. “everybody’s favorite food is pizza.” Now suppose you add vampires to your game, and so you change this to:

```
FavoriteFood[person, "blood"].If(Vampire[person]);
FavoriteFood[person, "pizza"].Fact();
```

This is a common coding idiom. You have a rule that expresses a general default at the end and then rules for exceptions and special cases before it. Since it runs the rules in order, it will check the vampire rule first. This code is correct in the sense that if you ask it what a vampire’s favorite food is:

```
FavoriteFood["Dracula", food].SolveFor(food)
```

It will return food=“blood”. And for most purposes, this is good enough. However, technically, the rules say that *both* blood *and* pizza are Dracula’s favorite foods; `FavoriteFood["Dracula", "pizza"]` will return true. If you want the rules to get that right, then you have to say:

```
FavoriteFood[person, "blood"].If(Vampire[person]);
FavoriteFood[person, "pizza"].If(!Vampire[person]);
```

NOT IS WONKY FOR GOALS WITH UNBOUND VARIABLES

The semantics of `!`*goal* (aka Not[*goal*]) are that it succeeds (is true) if *goal* is false (fails) and it’s false (fails) if *goal* is true (succeeds). It just literally runs *goal* to see if it can get a solution. If it can, it throws that solution away and fails. If *goal* fails, then `!` returns without binding any new variables.

This behaves pretty normally if goal doesn’t have any unbound variables. If we run `!Parent["Jayden", "Sora"]`, then it runs `Parent["Jayden", "Sora"]`. Let’s say that’s true and it succeeds. Then that means `!` fails. But if we run `!Parent["Jayden", "Jayden"]`, then it runs `Parent["Jayden", "Jayden"]`. That fails, since no one is their own parent. And so that means `!` succeeds.

But things are more complicated if we say `!Parent["Jayden", x]`. If x is already bound to some person, then this behaves like the example before. But if it’s unbound, we might expect it to mean “tell me an x who isn’t Jayden’s parent.” But it doesn’t mean that; it means “run `Parent["Jayden", x]` and tell me if it fails.” And when x is unbound, `Parent["Jayden", x]` means “find me an x who *is* Jayden’s parent,” and assuming Jayden has a parent, that goal will succeed, which means `!Parent["Jayden", x]` will fail. The query `!Parent["Jayden", x]` effectively means “Jayden has no parents ☹”

That might be fine except that, as we said, if x is bound we sort of get different behavior. And that means that if we have two goals, it matters quite a lot, which one comes first. If we say:

```
Person[x], !Parent["Jayden", x]
```

Then we really are asking for a person who isn’t Jayden’s parent, since the Person call will bind x before we get to the ! call. But the query with the subgoals reversed:

```
!Parent["Jayden", x], Person[x]
```

will fail because x starts out unbound and `!Parent["Jayden", x]` fails when x is unbound.

The general rule of thumb for using `!` is to **put your negations at the ends of your rules** to make sure everything is bound.

ADDING AN INTERACTIVE COMMAND LINE TO YOUR GAME

TELL has enough run-time information to convert a string into a query and run it. This is called a Read/Eval/Print/Loop. However, in order for the REPL to know that "Parent" means the `Parent` predicate, it has to have a list of all of the predicates the user should be able to call. This is stored in the `TELL.Program` class. To use the REPL, add:

```
var program = new Program("my program");
program.Begin();
```

before you start making predicates. Then call:

```
program.End();
```

when you're done. Then read a query string from the user and call:

```
program.Repl.Solutions(query)
```

This will return an `IEnumerable<object[]>` with the values of the variables of the *query* for each possible solution to the query.