The botl language

Botl is a logic programming language intended for game AI programming under Unity. Its design is heavily constrained by the requirement of being able to run with stack allocation only while co-existing with Unity's GC and object model. While the result will frustrate Prolog programmers,[1] it allows for a somewhat more novice-friendly language, and at least handles the use cases it was intended for.

Predicates, properties, and relations

A predicate is something that is either true or not true of its arguments. A predicate of only one argument represents some **property** that that argument either does or doesn't have. For example, a predicate called "npc" might tell you whether some object is an NPC, and the expression npc(joe) means Joe is an NPC. A predicate of two or more arguments represents some **relation** between its arguments. For example, the a predicate named "sibling" might express that two characters are siblings, and so sibling(joe, jane) means that Joe and Jane are siblings.

Queries

In logic programming, your programs are expressed as collections of statements about predicates, and "running" the program involves asking a **question** either about whether some predicate is true of some set of arguments, or of whether it's possible to **make it true** by filling in variables appropriately. For example, the query:

```
npc(joe)
```

asks the system "is Joe an NPC"? and the system will either respond true or false. However, you can also ask:

```
npc(X)
```

Here X is a variable (anything beginning with a capital letter or underscore is a variable, which is also why we haven't been capitalizing Joe's name). A query with a variable effectively asks the system to find values for the variables that make it true, in which case it will respond with both "True" and the value of X that makes it true (e.g. X=joe). The system can't answer all such questions, but it does a better job than one might expect.

The expressions npc(joe), npc(X), sibling(jane, X), are called **literals**: they give a predicate and its arguments. Literals can be combined into more complex queries using the logical connectives, "and', which is written as a comma, or, which is written as |, and not, which we will talk about later is it's one of the weak points of logic programming, and some would argue, of logic in general.

---

[1] In particular, it doesn't allow Prolog's "structures", meaning you can't have the unification process create and match data structures. That, in turn, also means you can't do metaprogramming. Prolog programmers, please believe me that I'm not any happier about this than you are. It's possible we'll back off of this limitation in the future, although it would make interoperation with C# somewhat more painful.

Thus a query can look something like:

```
npc(X), sibling(X, Y)
```

which means "find an X and Y where X is an NPC and Y is their sibling (NPC or otherwise)." When you have multiple literals in a query, the system tries to find one set of values for the variables that make both literals true.

You'll notice that this makes logic programming queries look a lot like database queries. And indeed, there are deep relationships between logic programming and databases. One difference, however, is that a query language like SQL isn't intended to be a general purpose programming language, while logic programming languages are designed for more general application.

Rules and tables

We haven't yet said anything about how the system determines the truth of queries. More specifically, we haven't told you how to tell the system about NPCs, siblings, etc. The answer is you tell it by defining predicates. Programming in a logic programming language is a process of specifying information about predicates. In Botl, there are three kinds of predicates

- A few built-in predicates
- Predicates explicitly defined by tables, as in a database
- Predicates defined by rules

We'll get to the built-in predicates later. Let's start with tables. You can define a predicate by making a CSV file containing the data for it, and then loading it into the system. For example, you can make a **spreadsheet**, sibling.csv:

Name1	**Name2**
joe	jane
bruce	betsy
ellie	beth

Now if you do a query sibling(X,Y), you'll get back X=joe, Y=jane. You can also query sibling(ellie, beth) and it will return True. Unfortunately, if you query sibling(beth, ellie), it will return false because the system doesn't understand symmetric predicates (although we might fix that in the future).

We can fix this with **rules**. Let's rename our spreadsheet and table from sibling to declared_as_siblings. Now we can define the real sibling predicate using the code:

```
sibling(X, Y) <-- declared_as_siblings(X, Y)
sibling(X, Y) <-- declared_as_siblings(Y, X)
```

Each rule specifies a situation in which the sibling relationship holds: it holds if X is declared as a sibling to Y, or if Y is declared as a sibling of X. When you ask the system if X and Y are siblings, it will check both conditions and report back if either is true. If neither is true (and no other sibling rules have been provided), then the query will **fail**, and the system will print False.

Value expressions and functions

Arguments to predicates may be arbitrary expressions. Unlike Prolog, these expressions are evaluated at the time of the call. So p(X+1) computes the value of X+1 and passes that to p, rather than passing the syntax tree for the expression X+1, as Prolog would do. A number of built-in functions, such as +, -, *, /, are built in.

Because these functional expressions are evaluated at call time, any variables appearing in them must already be instantiated. That is, if you say X+1, the system needs to already know the value of X. This is because to do otherwise would require the system be prepared to solve arbitrary algebraic expressions for the values of their variables. That's too expensive to appear as a programming language feature, and for sufficiently complicated examples, not even computable.

Predicates as functions

In addition, predicates can be declared to be functions using the function declaration:

function *name*/*arity*

Which states that the predicate with the specified *name* and *arity* (number of arguments) is also a function in the sense that its last argument can be thought of as an output determined by its other arguments (inputs). Such predicates can then be used freely in functional notation, rather than as explicit predicate calls.

For example, if one loads a table age/2, in which the first argument is a person and the second argument their age, this is conceptually a function: it maps persons to their ages. Normally, if you wanted to look up the age of a person, you would have to say age(Person, A) to get their age A. However, if you declare age/2 to be a function, you can then just say age(Person), as in age(Person) > 7. The system will automatically transform this into the conjunction: age(Person, A), A > 7. In addition, you can write expressions like age(Person) = 27 to mean age(Person, 27).

Note that this provides an exception to the rule that variables in functional expressions must be instantiated, since the predicate call gets hoisted from the functional expression and run in advance. So you can use a query like age(Person) > 7 to solve for persons who are older than 7.

Examples

Factorial

Factorial is a nice example of using the function declaration on a predicate to simplify its definition and calls. Factorial is a predicate: it has two arguments, the first of which is the number you're taking the factorial of, and the second is its factorial. And you can do standard predicate queries on it, such as asking if the factorial of a number is some other number:[2]

function factorial/2

[2] Although this definition won't work for asking factorial(X, 6), i.e. "what number has a factorial of 6"?

```
factorial(1)=1
factorial(N) = N*factorial(N-1) <-- N>1
```

This code is automatically transformed into the code:

```
factorial(1,1)
factorial(N, Temp1) <-- N>1, factorial(N-1, Temp2), Temp1=N*Temp2
```

Which forces the recursive call and the multiplication to happen in the right order.

Tree structures

This is a simple, standard tree recursion example. We assume the parent/child relation (the edges of the tree) is available as a predicate child(Parent, Child) and define other relations in terms of it. For simplicity, the code below assumes child/2 is a table, but it need not be.

```
table "child.csv"

sibling(X,Y) <-- child(P, X), child(P, Y)

descendant(X,X)
descendant(X, Z) <-- child(X, Y), descendant(Y, Z)
```

Graph structures

Handling graphs is more complicated because of the issue of cycles. Here we use a C# hashset to track the set of notes that have already been searched:

```
table "graph.csv"

adjacent(X,Y) <-- graph(X,Y)
adjacent(X,Y) <-- graph(Y,Z)

connected(X,Y) <-- Visited = hashset(), connected(X,Y, Visited)

connected(X,Y, _) <--adjacent(X,Y)
connected(X, Y, V) <-- adjacent(X, Z), not(Z in V), V.Add(Z), connected(Z, Y, V)
```

C# interop

Suppose you have a class, Thingy, with a static field ThingyTable that is a Dictionary<string, Thingy> holding all the instantiated Thingies, indexed by their names. Suppose you want to find a Thingy whose Squishy property is true. Then all you'd have to do is say:

```
Pair in $Thingy.ThingyTable, Pair.Value.Squishy
```

(Recall that Dictionaries a collections of KeyValuePairs, so when you use in/2 it iterate over the dictionary, you're getting back KeyValuePairs, whose Key field is the string and whose Value field is the Thingy.)

Choosing a behavior

A simple example of using arg_max to do optimization. Arg_max/4 is a built-in predicate that can be called as a function. It enumerates the solutions to a query, finds the highest scoring one and returns the value of a specified variable in the highest scoring solution:

possible(behavior1) <-- … *whatever* …
possible(behavior2) <-- … *whatever* …
…

function utility/2
utility(behavior1) = *whatever* <-- *whatever*
utility(behavior1) = *whatever* <-- *whatever*
…

chosen_behavior(Best) <-- Best = arg_max(utility(B), B, possible(B))

A dumb reactive planner

Simple example of using structs to reifying actions and predicates to interpret a reactive planner language. This is one case where one really misses full Prolog, which would allow much more flexible :

```
// Type declarations
struct action(Type, Actor, Patient, Arg1, Arg2)
struct state(Predicate, Arg1, Arg2)

signature achieve(state, action)
signature runnable(action)
signature precondition(action, state)
signature postcondition(action, state)
signature holds(state)

// These three lines do the backward chaining for the planner.
achieve(Goal, nop) <--   holds(Goal), !
achieve(Goal, A) <-- postcondition(A, Goal), runnable(A)

runnable(A) <-- precondition(A, S), holds(S)

// Very ugly; this is where you wish you had read prolog.
holds(state(P, Pad1, Pad2)) <-- missing(Pad1), missing(Pad2), !, call(P)
holds(state(P, A, Pad)) <-- missing(Pad), !, call(P, A)
holds(state(P, A, B)) <-- call(P, A, B)

// Formalization of give action
precondition(give(Giver, Object, _Givee), have(Giver, Object))
postcondition(give(Giver, Object, Givee), have(Givee, Object))

// Trivial knowledge base
have(fred, apple)
```