Reference manual

Comments

Comments are C++-style, i.e. // or /* */

Value expressions

A **value expression** is a constant, variable, or function call (whose arguments are themselves value expressions). Value expressions have more or less the same syntax and semantics as other languages.

Constants

Constants can be

- **Integers** (1, 10, -5)
- Single-precision **floats** (5.0, -3.2)
- **Booleans** (true, false)
- **Strings** (e.g. "this is a string")
- **Symbols**
  Symbols are essentially strings that don't require quotes. A symbol is any sequence of letters and digits that starts with a lower-case letter, or a sequence of operator characters (e.g. =, <, <=, +, and so on).

  NB: Symbols are interned in a hash table, so they can be compared in constant time, whereas strings require linear time.

Named entities

There is a general mechanism for referring to other kinds of objects that have names. These can be referred to by saying $*name*. The named entities currently supported are:

- $*globalVariable*
  Denotes a global variable (see below) such as $time, $this, or $gameobject.
- $*typeName*
  Denotes a the type object for one of the underlying CLR/mono/Unity types. So $String denotes the CLR/mono's string datatype, $GameObject denotes Unity's GameObject datatype, and $RigidBody2D denotes the Unity RigidBody2D datatype.
- $*gameObjectName*
  Denotes the Unity gameObject with the specified name. Note that this is resolved at compile time for efficiency, so it can only be used to access objects that exist at compile time. For other objects, call the GameObject.Find(string name) method directly.

Variables

Following Prolog and ASP, variables can be named as any sequence of letters, digits, or underscore (_) characters but must start with either an upper case letter, or an underscore. Thus `Foo`, `FOO`, `_Foo`, and `_foo` are variables, but plain `foo` is a symbol.

Variables are "logic variables", i.e. they can only be assigned once, but they can be aliased to other variables so that assigning one of them effectively assigns both of them.

Global variables

These implement the equivalent of global variables in other programming languages. They can be read by saying $*name*, and can be updated using set_global(*name*, *newValue*). Note that the *name* in set_global should not be prefixed with a $. Three globals are built in and automatically set when C# calls into BotL using the IsTrue() methods.

- `$this`
  Set to the Unity Component on which the IsTrue() method was called.
- `$gameobject`
  Set to the GameObject on which the IsTrue() method was called, or if IsTrue() was called on a Component, then on its GameObject
- `$time`
  Set to the value of Unity's Time.time

In addition, you can create new global variables using the declaration:

> `global` *name*=*initialValue*

Note that this must appear before the first use of the variable and that the *name* should not be prefixed with a $.

Function calls

A function call looks like function calls in other languages, and has similar semantics: `f(1)`, `f(g(X), 1)`, etc. As with Prolog, certain functions can be declared to be prefix or infix operators. Thus `1+1` really means `+(1,1)`. Arguments to functions may be arbitrary value expressions.

Member references

Member references are notated with an infix dot, as in other languages: a.fieldName, b.MethodName(1, X), etc. They have the same semantics as other languages. Arguments to member references may be arbitrary value expressions.

NB: evaluation of member references goes through reflection, and so allocates memory that must later be garbage collected. Most other operations in Botl use stack allocation and so don't place pressure on the GC.

Predicates

A predicate represents a relation between sets of values. The predicate is true iff the relation holds between the arguments to the predicate. For example, the = relation is true precisely when its arguments are identical. The < predicate is true when its first argument is strictly less than its second.

Names of predicates follow the same rules as symbols. That is, f, foo, +, and < are valid predicate names, but not Foo.

Literals

A **literal** is a call to a predicate. It contains the name of the predicate to call, together with a list of its arguments, which are value expressions:

name(*arg*$_1$, … *arg*$_n$)

If the predicate has no arguments, then a call to it is simply the name of the predicate, without parens. The arguments to the literal can be any functional expression.

Thus the following are valid literals:

- foo
- bar(1)
- bar(1+1)
- baz(1, X)
- baz(f(1), X*2)

NB for Prolog programmers: unlike Prolog, the value expressions for the arguments to a predicate are evaluated before the predicate is invoked. Thus `p(f(X))` is equivalent to the Prolog expression: `Y is f(X), p(Y).`

Operators

Some binary predicates, such as =, <, in, etc., can be written in infix notation. So the literal X = Y is exactly equivalent to the literal =(X, Y). In addition, some unary predicates, such as not, can be written as prefix operators without parentheses. That is, not p(X) is exactly equivalent to not(p(X)).

Source file format

Source code is a series of declarations and predicate definitions, possibly with comments interspersed.

Defining predicates

Predicates are defined by a series of **clauses** that specify when the predicate is true. For example:

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

states that Y is a descendant of X if they're the same individual or if Y is a descendant of one of X's children.

Clauses are generally **rules** of the form:

goal <-- *subgoal*

where *goal* is a literal and *subgoal* is an arbitrary query. The rule states that when calling a predicate whose arguments match the *goal* literal, the system can treat it as instead being the query *subgoal*. If the subgoal succeeds, then the original query succeeds.

Alternatively, a clause can simply state that a goal is always true. In this case, the <-- and subgoal are omitted.

Declarations

Most declarations inform the system about special properties of particular predicates. Predicates are specified using **predicate indicators** of the form *name*/*arity*. Thus location/2 refers to the 2-argument predicate named location. Predicates with the same name but different numbers of arguments are allowed, but are considered distinct predicates.

- `require` *name*
  Tells the system to load the named source file unless it has already been loaded. Use this to ensure that a given file's dependencies (e.g. function, struct, and signature declarations) have already been loaded.

- `table` *name*/*arity*
  Tells the system to create a new table with the specified name and arity.

- `function` *name*/*arity*
  Tells the system that the specified predicate is considered a function and so can be embedded in functional expressions, the last argument is always considered to be the "output" of the function. Thus, if `f/2` is declared to be a function, the system will allow Y=`f(X)` as syntactic sugar for `f(X,Y)`. More generally, occurances of f with 1 argument in the arguments to other predicates will be hoist out. Thus `p(f(X))` will be transformed into `f(X,Y), p(Y)`.

- `struct` *typeName*`(`*slots …*`)`
  Tells the system that the specified *typeName* is a structure consisting of the specified slots. For example, `struct condition(predicate, arg1, arg2)` says that a `condition` argument consists of three slots: a `predicate`, and `arg1` and `arg2`.

- `signature` *predicate*`(`*argumentTypes …*`)`
  Used when *predicate* takes at least one struct as an argument. Tells the system that when *predicate* is called with the same number of arguments as given in the signature, those arguments should be expanded according to the types given. For example, the declaration `signature believes(object, condition)` tells the system that when `believes` is called with two arguments, the second is really a `condition`, and so should be expanded into

3 actual arguments: the predicate, arg1, and arg2. This, a call `believes(fred, loves(jenny, fred))` would expand into `believes(fred, loves, jenny, fred)`.

Defining tables using CSV files

CSV files can be used to specify a table. The CSV file should consist of a header row, followed by a row for each entry in the table. The rows should have one column per argument to the predicate. Columns are assumed to be strings, unless their respective headers end with the notations `(int)`, `(float)`, or `(symbol)`, in which case the cells for that column will be converted appropriately. Thus the CSV file:

Person	Age (int)
joe	10
jane	11

Would make a table with two rows, the columns of which are strings (with entries “joe” and “jane”) and numbers, respectively, while:

Person (symbol)	Age (int)
Joe	10
Jane	11

Is the same, but would treat the first column as symbols instead of strings.

Built-in predicates

Trivial predicates

- **fail**
  Always false.
- **true**
  Always true.

Equality and inequality

- X=Y
  True when its arguments are identical.
- X\=Y
  True when X and Y are different.
- X<Y, X=<Y, X>Y, X>=Y
  True when arguments are numbers with the specified relationship.

Type testing

- **integer**(X), **float**(X), **number**(X), **string**(X), **symbol**(X)
  True if X has the specified type.

Binding testing

- **var**(X)
  True when X is an uninstantiated variable, i.e. a variable that hasn't been given a value.
- **nonvar**(X)
  True when X isn't an uninstantiated variable

Metapredicates

- **not** *Query*
  Runs *Query* and . If *Query* succeeds, not *Query* fails, and if *Query* fails, not *Query* succeeds.
- **forall**(*Generator*, *Condition*)
  True if for every variable binding generated by the query *Generator*, *Condition* is also true.
- **call**(*PredicateName, Args …*)
  True if PredicateName(Args …) is true.
- **min**(Score, Generator, Result)
  max(Score, Generator, Result)
  Functions. Result is the minimal value of Score over all solutions to Generator.
- **arg_min**(Score, Arg, Generator, Result)
  arg_max(Score, Arg, Generator, Result)
  Functions. Result is the value of Arg corresponding to the minimal value of Score over all solutions to Generator.
- **sum**(Score, Generator, Result)
  Function. Result is the sum of Score over all solutions to Generator. Note that if Generator produces duplicate solutions, then they will be added repeatedly.
- **setof**(Element**:**Generator, Set)
  Finds the value of Element from all solutions to Generator, and returns a Hashset<object> of all solutions. Also callable as functional expression. Warning: this operation allocates memory in the C# heap.
- **listof**(Element**:**Generator, Set)
  Finds the value of Element from all solutions to Generator, and returns an ArrayList of all solutions. Also callable as functional expression. Warning: this operation allocates memory in the C# heap.

Control flow

- **!**
  Prunes choicepoints back to the state before the before the call to the surrounding predicate. Thus, any attempt to backtrack over a ! will fail the surrounding predicate, or disjunction. Note that this is a different semantics than Prolog. Botl treats a disjunction (i.e. or, the | operator) as a call to a new, anonymous predicate. So ! commits the current disjunction, not the entire surrounding predicate.
- *P* **->** *Q* **|** *R*
  True if *P* and *Q* are true, or if *P* is false and *R* is true. Equivalent to (*P*, !, *Q*) | *R*.
- **once**(*Query*)
  True when *Query* succeeds, but will not restart. Equivalent to (*Query*, !) | fail.

- **ignore**(*Query*)
  Runs *Query* and always succeeds. Equivalent to (*Query*, !) | true.

Table update

- **assert**(table(args, …))
  Adds (args, …) to table. All args must be instantiated.
- **retract**(table(args, …))
  Removes (args, …) from the table. All args must be instantiated.
- **retractall**(table(args, …))
  Removes all rows from table matching (args, …).
- **set**(function(args, …) = value)
  set(function(args, …) **+=** value)
  set(function(args, …) **-=** value)
  Updates value of a table which is declared to be a function.

C# interoperation

- X **in** Y
  True when Y is an IEnumerable and X is an element of Y.
- **item**(ilist, index, element)
  True when element = ilist[index]. Also callable as functional expression item(ilist, index) or ilist[index].
- **adjoin**(collection, element)
  Imperative. Adds element to collection.

IO

- **write**(X)
  Prints X to the console.
- **writenl**(X)
  Prints X to the console and then starts a new line

Loading code

- **load**(Path)
  Loads a .bot file.
- **load_table**(Path)
  loads a CSV file into a table. The table is given the same name as the file name portion of Path (without extension).

Functions and operators for value expressions

Arithmetic

- X**+**Y, X**-**Y, X*****Y, X**/**Y, **-**X
  Standard arithmetic operators.

- **sum**(Score, Generator)
  The sum of Score from all solutions to Generator. If Generator produces duplicate solutions will be multiply summed. Expanded into a call to sum/3.
- **min**(Score, Generator)
  max(Score, Generator)
  The smallest/largest value of Score from all solutions to Generator. Expanded into a call to min/3 or max/3, respectively.
- **arg_min**(Score, Arg, Generator)
  arg_max(Score, Arg, Generator)
  The value of Arg corresponding to the smallest/largest value of Score from all solutions to Generator. Expanded into a call to arg_min/3 or arg_max/3, respectively.
- **setof**(Element**:**Generator)
  Returns a hashset of values of Element for all solutions to Generator.
- **listof**(Element**:**Generator)
  Returns an ArrayList of values of Element for all solutions to Generator.

C# interop

- Object**.**FieldName
  Returns the value of the specified field of the specified object.
- $Type**.**FieldName
  Returns the value of the specified static field.
- *GameObject*`::`*ComponentType*
  Returns the specified component of the specified *GameObject*. For example, `$foo::RigidBody2D` returns the RigidBody2D of the game object named "foo".
- Object**.**Method(args)
  Calls the specified method on the specified object with the specified arguments and returns its result.
- $Type**.**Method(args)
  Calls the specified static method and returns its result.
- **new** Type(args)
  Creates a new C# object of the specified type.
- List[Index]
  Returns the specified element of the list.

Other

- **array**(elements, …)
  Returns an object[] array containing the specified elements.
- **arraylist**(elements, …)
  Returns an ArrayList containing the specified elements.
- **hashset**(elements, …)
  Returns a Hashset<object> array containing the specified elements.

More to come, obviously…

Features that would be good to add but have not yet been added

- More math operations
- Make more Unity builtins available in functional expressions
- Language support for pooling of data objects
- Language support for symmetric and/or transitive tables
- Support for tables whose rows contain variables
- Support for eremic logic