Step Language Reference

Table of Contents

Overview

This manual intended for readers already familiar with *Step* or at least other programming languages. Those who are new to *Step* are encouraged to start with the *Step* tutorial.

Step is a highly expressive, declarative language for text generation. Semantically, it can be seen as a kind of logic programming language with the addition of a range of imperative commands (e.g. text output, assignment statements) that are automatically undone upon backtracking. However, syntactically, it tries to allow simple code, and especially code that is just trying to print text, to look as much like normal text as possible. The overall goal is to make a formally powerful language where writers can start simply and only learning those features they need, or for multideveloper projects, AI programmers can do very sophisticated work which is hidden away in its own files, while writers are free to write narrative with minimal intrusion of code.

Tasks and methods

A Step program consists of a set of **tasks**, which are the equivalent of subroutines in another language, or predicates in a logic programming language. The code for a task is defined by a set of rules, called **methods** for strategies to try when the method is called. Methods define a **pattern of arguments**, providing either variables or specific values for each one, and code to execute when the arguments in the call **match** the pattern specified in the method.

Nondeterminism

Step is a nondeterministic language, meaning that a given call may underdetermine which method will be executed to achieve it. Since a given call might match several methods, and each of those methods might contain other calls that each match multiple methods, a given call might correspond to many different **execution paths**. However, calls and methods are allowed to **fail** if, for example, a call has no methods that match it. Failure is not necessarily an error condition. Instead, signals Step to try the next method, whatever it might be. This is called **backtracking**, and it causes any actions performed by the failing method to be **undone** so that when the next method is tried, it is as if the failed method(s) were never executed. If all candidate methods for a call fail, the call itself fails and the method calling it backtracks.

Nondeterminism is useful in a number of ways. For one, it leads to a natural style of programming in which one can simply provide many different ways of trying to perform the different tasks, not all of which will work for a given call, and trust the system to find a way of making it work if there is one. For another, it makes it very easy to express certain kinds of logical inference (hence the term “logic programming”) and certain kinds of generative planning problems. More generally, it lets you solve search problems declaratively, provided the problems are solvable using a naïve depth-first-search strategy.

Pattern-directed invocation

Methods are chosen for a call by matching the arguments provided in the call to the argument pattern specified in the method. Arguments in a call or method pattern can be variables, which act as wildcards, constants (numbers, strings), or tuples. The rules for matching (aka unification) are that two values match if

- They are the same constant
- One is a variable, in which case a note is made that the variable's value is henceforth the other value
- They are both tuples and their respective elements unify

If variable has been given a value, that value is used for the matching rules above; the system will not attempt to give the variable a new value. When two different variables, neither of which have values, are matched, the system remembers that in future, those two variables must have the same value.

If argument matching fails partway through, or if a method is backtracked, then any variable assignments performed as part of the matching are undone.

Randomization

By default, the system tries methods in the order they appear in the source file. However, by tagging a task `[randomized]`, the system will try its methods in a different random order each time it is called. The relative frequency of different methods can also be adjusted.

Text vs. code mode

Step uses two different syntactic conventions. Calls to tasks and argument patterns in methods are written in **code mode**: they distinguish between types of tokens based on their first characters: capitalized words are assumed to be references to global variables, lowercase words are string constants, and tokens beginning with a `?` are assumed to be local variable references. However, method bodies are written in **text mode**: all tokens are treated as verbatim text to be printed, save for variable substitutions and code in square brackets. This makes it easy to write methods that just print things rather than having to write them as explicit calls to the `Write` primitive along with a quoted string.

Expression syntax in code mode

Expressions are used to notate tasks to call or arguments passed between them. An expression is one of the following:

- Constants
    - **Number** (`1`, `-4.5`, *etc*.)
      The C# types System.Int32 and System.Single.
    - **Boolean** (`true` or `false`)
      The C# type System.Bool
    - The **null value** (`null`)
    - **String** constant (`foo`, `bar`, `fooBar`, `foo\ bar`, `|this is a string constant|`)
      Any word starting with a lower-case letter. A backslash can be used to escape spaces, and vertical bars can be used to quote strings. This corresponds to the C# type System.String.
    - **Text** (`"this is some quoted text"`)
      You probably don't need to know this, but double quotes denote tokenized text, not strings. Text is internally represented in tokenized form so that for the final generation of the string output, it can reason about things like capitalization. But this does mean that the C# type of

quoted text is an array of strings, not a single string, and that `"x"` and `x` are non-equal expressions.

- **Tuple** (`[this is [a tuple]]`)
  A tuple is a series of expressions enclosed in square brackets. Its C# type is object[], i.e. an array of objects. It is not a linked list.

- Variables
  Variables are sequences of alphanumeric characters, possibly including ? and _, and are distinguished from string constants by their first character: if the first character is a ?, it is a local variable, if it is capitalized, it is a global variable. Otherwise, it is a string constant.
  - **Anonymous local variable** (`?`)
    A bare question mark is used to denote a fresh local variable. Each use of a bare question mark denotes a *different* local variable. They are used to denote wildcards where a value should be ignored completely.
  - **Local variable** (`?`, `?x`, `?FooBar`, `?_foo_bar`, etc.)
    Local variables are visible only within their own methods. They begin unbound (having no value) and acquire values through unification (matching to values in calls).
  - **Global variable** (Task, X, CamelCase)
    Global variables are visible to all methods. They are given values either through method declarations (when the variables denote tasks) or a `[set` *Variable* `=` *expression*] command. Unification will not assign them values, and they may not be set to unbound local variables.

Tasks arguments are both input and output

Tasks do not have a return value in the sense of languages like Python or C#. Instead, a caller passes a value as input by passing either a constant or a variable with a value. It receives an output by passing an unbound variable and allowing the called task to give a value through unification.

For example, we can represent a directed graph, using a task `Edge` that will succeed when its arguments are vertices and there is an edge in the graph from the first argument to the second. We can then represent the graph by writing one method per edge:

```
[predicate]
Edge a b.
Edge b c.
Edge c d.
Edge b x.
Edge x y.
Edge m n.
```

We can now ask if two vertices are adjacent by calling Edge with two constants: `[Edge a b]` will succeed (is true), while `[Edge c a]` will fail (is false). We can ask what vertex a is adjacent using `[Edge a ?x]`, which will bind the variable `?x` to b. Conversely, `[Edge ?x b]`, which will bind the variable `?x` to a. And the call `[Edge ?x ?y]` asks for any edge. Thus, Edge's arguments can each be either inputs or outputs, depending on the context. We can then use this to implement reachability analysis for DAGs very compactly:

```
[Predicate]

# vertices are reachable from themselves
Reachable ?v ?v.
```

```
# e is reachable from s if it's reachable from an adjacent vertex i
Reachable ?s ?e: [Edge ?s ?i] [Reachable ?i ?e]
```

As before, this predicate can treat it's arguments as inputs or outputs as the caller desires.

Structure of a Step program

A Step program consists of a set of files in a directory, typically either Documents/Step/*projectname* or Documents/Github/*projectname*. Step loads all `.step` and `.csv` files in the directory. The order of loading is undefined.

CSV files

A CSV file is assumed to define a Step task with the same name as the file. This allows tabular data to be imported relatively easily. The first row is treated as a header. Subsequent rows are interpreted as methods for the task, with the columns being expressions for the arguments of the method.

By default, a column is treated as an argument to the predicate. However, if the header for the column ends with a `?`, it defines a separate predicate over the values in the first column, being true of the value in the first column whenever the new column says "true" or "yes". If the name in the header begins with a `@`, the column defines a binary predicate where the first argument is the value in the first column and the second argument is the value in this column. For example, the CSV file, assumed to be in the file `Monster.csv`:

Monster	@PluralForm	Singular?
reptoid	reptoids	yes
hungry dead	hungry dead	no
vampire	vampires	yes
wizard	wizards	yes
werewolf	werewolves	yes
demon	demons	yes

defines the following methods:

```
Monster reptoid.       PluralForm reptoid reptoids.         Singular reptoid.
Monster hungry_dead.   PluralForm hungry_dead hungry_dead.
Monster vampire.       PluralForm vampire vampires.         Singular vampire.
Monster wizard.        PluralForm wizard wizards.           Singular wizard.
Monster werewolf.      PluralForm werewolf werewolves.      Singular werewolf.
Monster demon.         PluralForm demon demons.             Singular demon.
```

Method weights can be specified by naming the first column "`[#]`" and placing the weight for a row in that column. Blank weights will be treated as the default weight (1).

Note that the CSV input format doesn't allow task annotations. So assigning attributes, such as *`[randomly]`* would require a separate declaration within a `.step` file, e.g.:

```
[randomly]
predicate MyCSVPredicate ?col1 ?col2 ?col3.
```

See **Task and predicate declarations**, below, for more information.

Step files

A `.step` file contains a series of declarations (see below), possibly mixed with comments. The rest of this document will focus on the manual of such declarations, since CSV files are relatively simple.

Comments

Comments in `.step` files are notated with a #.

Declarations

A `.step` file consists of a series of declarations, possibly with comments interspersed.

Method declarations

The vast majority of a `.step` file consists of a set method declarations. A method consists of a Head (the task name and argument pattern) and a *Body* (the text to print and/or subtasks to call), and optionally a set of *annotations*, which can assign special properties to the task such as randomization or being treated as a predicate, or to the individual method, such as changing its weight for rantomization.

These take one of the following forms:

- [*annotations*] *Head* **:** *Body*
  When the *Body* is a single line, the head and body are separated with a colon. The annotations need not be on the same line as the head, however.
- [*annotations*] *Head* **.**
  When there is no *Body* to execute, the method is terminated with a period. Again, the annotations need not be on the same line.
- [*annotations*] *Head* **:**
  Body
  [end]
  When the *Body* is multiple lines, the first line consists of the *Head* and colon, and the body is terminated by `[end]`. As always, the annotations, in included, need not be on the same line as the *Head*.

Head format

The is a pattern to match to a possible call. It consists of the name of the taskand a series of expressions separated by spaces:

> *TaskName argument* …

The *TaskName* must be a global variable name, and so must be capitalized. The *argument* expressions may be:

- A normal expression (see *Expression syntax in code mode*)
- (*Predicate ?localVariable*)
  A parenthesized expression denotes the *?localVariable*, but requires that the *Predicate* (task) be called on it and succeed in order for the method to match. The method:

  ```
  SomeTask (Var ?x): My argument is a variable!
  ```

  is syntactic sugar for:

```
SomeTask ?x: [Var ?x] My argument is a variable!
```

Body format

The *Body* of a method declaration is written in text mode, meaning that any tokens in it are, by default, printed verbatim. However, a number of constructs are handled specially. All involve either text enclosed in square brackets, or prefixed by a ? or ^:

- [*Var args* …]
  Square bracketed calls to subtasks are run and their output substituted into the body. If the call fails, the system backtracks to the previous call in the body, or if this is the first call, the method fails.
- *?localVariable* or ^*GlobalVariable*
  Local variable names or global variable names preceded by an ^ character indicate a variable substitution (see below).

Variable assignments and arithmetic expressions

These modify the value of a global variable. All are undone upon backtracking.

- [`set` *variable* = *expression*]
  Sets *variable* to the value of *expression.* The expression may be an arithmetic expression, in which case the variable is given the value of the arithmetic expression, and so may be several tokens long. This is the only construct that allows arithmetic expressions.
- [`inc` *GlobalVariable expression*]
  [`inc` *GlobalVariable*]
  [`dec` *GlobalVariable expression*]
  [`dec` *GlobalVariable*]
  Adds/subtracts *expression* from *GlobalVariable*. If no expression is given, adds/subtracts 1.
- [`add` *expression GlobalVariable*]
  Pushes the value *expression* onto the beginning of the list stored in *GlobalVariable*.
- [`removeNext` *variable GlobalVariable*]
  Removes the next element of the list *GlobalVariable* and bdings *variable* to it.

Fluent update

These are undone upon backtracking.

- [`now` *updates* …]
  Updates one or more fluents (dynamic predicate). The updates must be of the forms [*Fluent args* …] or [`Not` [*Fluent args* …]]. The former tells the system the fluent is now true of the specified arguments, the latter that it is now false of those arguments. The arguments must all be ground; that is, they may not be unbound variables.
- [`now` *variable = expression*]
  Synonym for [`set` *variable = expression*], which sometimes reads more naturally in context.

Branching and sequencing

These use different mechanisms to choose one of the enclosed *bodies*. They are simply convenience features to save you having to write separate tasks and methods.

- `[randomly]` *body* `[or]` *body* `[or]` … `[end]`
  This randomly selects one of the *bodies*. If it fails or if the system backtracks for some other reason, it will try each of the *bodies* once until it finds one that works. If none work, it fails.
- `[firstOf]` *body* `[or]` *body* `[or]` … `[end]`
  Runs the bodies until it finds one that succeeds. If backtracked, it will pick up with the next one. If none work, it fails.
- `[sequence]` *body* `[or]` *body* `[or]` … `[end]`
  The first *body* the first time this method is called, the second *body* the second time it is called, etc. Will not try successive *bodies* if backtracked, and will fail entirely if the method is called more times that there are *bodies*.
- `[case` *expression*`]` *Predicate* `:` *body* `[or]` *Predicate* `:` *body* `[or]` … `[else]` *body* `[end]`
  Equivalent to `[firstOf]` `[`*Predicate expression*`]` *body* `[or]` `[`*Predicate expression*`]` *body* `[or]` … `[or]` *body* `[end]`.
- `[cool` *count*`]`
  Implements a "cooldown". Allows execution in this branch of the body once, then fails for the next *count* executions of this method. Used to mark text that can be used, and used repeatedly, but should not be used twice in quick succession. If backtracked, its effect is undone.
- [once]
  Marks this branch as executable only once. Equivalent to `[cool` *maxint*`]`.

English verb conjugation

This is a stopgap measure that will likely be replaced by a more general mechanism in the future. If so, we'll ensure backward compatibility.

- `[s]` or `[es]`
  Used after an English present tense verb. Attempts to conjugate the verb depending on the value of the global variable `ThirdPersonSingular`. The intent is that the variable would be set by `Mention`.

Variable substitutions

In text mode, a local variable is replaced with a call to `Mention`, which by default prints the value of the variable and so has the effect of substituting the value of the variable into the text. Since local variables begin with ? and I am not aware of any human language in which words or clauses begin with a ?, there is no particular ambiguity here. However, Global variables just look like capitalized words. So to substitute a global variable, you must prefix it with an up arrow (^):

- *text ?localVariable text*
  Equivalent to: *text* `[Mention` *?localVariable*`]` *text*
- *text ^GlobalVariable text*
  Equivalent to: *text* `[Mention` *GlobalVariable*`]` *text*

The default task to call is `Mention`, but this can be overridden by following the variable with a slash and the name of the task to call:

- *text ?localVariable/Task text*
  Equivalent to: *text* `[`*Task ?localVariable*`]` *text*
- *text ^GlobalVariable/Task text*
  Equivalent to: *text* `[`*Task GlobalVariable*`]` *text*

A + can be used to call multiple tasks on the variable:

- *text ?localVariable/Task1+Task2 text*
  Equivalent to: *text* [*Task1 ?localVariable*] [*Task2 ?localVariable*] *text*
- *text ^GlobalVariable/Task+Task2 text*
  Equivalent to: *text* [*Task1 GlobalVariable*] [*Task2 GlobalVariable*] *text*

Slashes can be iterated to look up the values of binary relations:

- *text ?localVariable/Relation/Task text*
  Equivalent to: *text* [*Relation ?localVariable ?tempVariable*] [*Task ?tempVariable*] *text*
- *text ^GlobalVariable/Relation/Task text*
  Equivalent to: *text* [*Relation GlobalVariable ?tempVariable*] [*Task ?tempVariable*] *text*

The slash can be iterated as many times as desired, and the + can used at the end. Thus:

```
?who/Brother/FirstName+LastName
```

is equivalent to:

```
[Brother ?who ?temp] [GivenName ?temp][FamilyName ?temp]
```

Here `Brother` is assumed to be defined so that the second argument is the brother of the first argument, and `GivenName` and `FamilyName` are assumed to print the given and family names of their arguments, respectively.

Annotations

Method annotations declare special properties of a method, or of the task to which it belongs. Annotations are always enclosed in square brackets. Multiple annotations can appear before the same method, in which case they must each be enclosed in brackets.

Task annotations

Although generally appearing in one particular method, task annotations declare a property of a task in general. The task thus has that property if any method has that annotation. They are therefore "sticky."

Randomization

- `[randomized]`
  The task should perform a weighted random shuffle of its methods when called. Although each call uses a different shuffle, the call remembers it's shuffle and is unaffected by any shuffling done by subsequent calls to the task.

Search control

By default, a call is run only once; if the system attempts to backtrack over the call, the call isn't restarted. In addition, if the call fails entirely, this is an error. This is different from a logic programming language. Logic programming semantics are enabled using the `[predicate]` annotation.

- `[predicate]` or `[generator]`
  States that it is not an error for the task to fail, and that the task should be retried if later code backtracks. This gives the task the semantics you would normally find in a language like Prolog. This is equivalent to: `[fallible][retriable]`.
- `[fallible]`
  States that it is not an error for the task to fail.

- `[retriable]`
  States that the call should be retried for a different solution if the code appearing after the call backtracks.

Stateful predicates

- `[remembered]`
  States that when the task is called, its results (arguments and text output) should be cached. Subsequent calls should be unified with the remembered arguments; when a match is found the remembered text, if any, should be printed and the task should complete.
- `[fluent]`
  States that the task is a `[predicate]` and, additionally, can be updated using `now`. Methods can still be declared for the predicate, in which case they behave as defaults.
- `[function]`
  For `[fluents]`, states that the predicate has the additional property of being a function in the sense that its last argument is unique given its previous arguments. This means that an update like `[now [F 1 2]]` will not only mark `[F 1 2]` as true, but mark any previous `[F 1 ?]` as false.

Other

- `[main]`
  States that the task is an entry point to the program and so the interpreter should suppress any warnings that it isn't called by other tasks.
- `[suffix]`
  States that this is a task that is allowed the rewrite the last token in the output. This feature is likely to be deprecated.

Method annotations

- [*number*]
  For randomized tasks. States that number should be used as the weight of this method for the weighted shuffle. Large numbers are proportionally more likely to be chosen first in the shuffle. The default weight is 1. In addition to controlling the sampling of the output, this can be used as a debugging tool. Prefixing a method with [0] effectively comments it out. Prefixing it with [999] effectively forces it to be tried first every time.

Task and predicate declarations

Task declarations provide a mechanism for declaring the existence, arguments, and optionally, annotations of a task without declaring methods for it. They are useful for declaring the properties of tasks defined in CSV files.

- [*annotations*] `task` *Head*`.`
  Defines the task, its number of arguments, and optionally annotations. However, it does not add any methods, whereas simply saying "*Head*`.`" would constitute a method for the task.
- [*annotations*] `predicate` *Head*`.`
  Defines the task, its number of arguments, and optionally annotations. Implicitly marks it `[predicate]`.
- [*annotations*] `fluent` *Head*`.`
  Defines the task, its number of arguments, and optionally annotations. Implicitly marks it `[fluent]`.

Global variable initialization

Global variables whose values are not task objects can be initialized using the pseudo-method `initially` (note lower case):

`initially: [set` *Global* `=` *value*`]` …

Any code in an `initially` method is executed at load time and the resulting global variable values are saved as the initial values to use in the future.

Built-in primitive tasks

Step contains a large number of built-in tasks that can be called, which are documented in a separate file that is automatically generated as part of the build process. However a few specifics should be mentioned here.

Printing

These are the basic operations used for printing text.

- `[Write` *object*`]`
  Prints *object*. If object is a string containing underscores, the underscores are converted to spaces.
- `[WriteVerbatim` *object*`]`
  Prints *object* without converting underscores are converted to spaces.
- `[WriteCapitalized` *object*`]`
  Prints *object*, capitalizing the first character. If object is a string containing underscores, the underscores are converted to spaces.
- `[Mention` *object*`]`
  A user-defined task called when text uses Variable substitutions. If no implementation is provided, `Write` is used.

Objects that can be treated as tasks

When there is an obvious interpretation of a native object as a predicate, the *Step* interpreter allows that object to be used as a predicate. In particular, the following types can be treated as predicates:

- `IEnumerable`
  Objects implementing the `IEnumerable` interface behave as unary predicates. Thus collections, including tuples and other arrays, can be "called" with an argument. If the argument is a variable, it is set to the first item of the collection, and subsequent backtracking will enumerate the elements of the collection, failing when the collection runs out of items. If the argument is not a variable, then it is treated as a test for membership in the collection. If the item appears in the collection, the call succeeds, otherwise it fails.
- `IDictionary`
  Objects implementing the dictionary interface are treated as binary predicates with the call `[`*dict key value*`]` being true when *key*'s value within *dict* is *value*.
- `bool`
  Booleans are treated as nullary (argumentless) predicates. `[true]` will always succeed once, and `[false]` will always fail.
- Text (`string[]`)
  Text is treated as a nullary (argumentless) task that prints itself.

C# interface

You can use Step in a C#/.NET project, such as a Unity game by adding the `Step.dll` library, creating a `Module`, loading files or directories into it, and then calling the tasks within the module.

Modules

A module holds a set of global variables and their initial values. Since tasks are stored in global variables, the initial, and generally final, value of a task variable is a task object containing whatever methods have been defined for it. Modules are implemented by the `Step.Module` class.

- `Module.Global`
  The module containing all built-in Step tasks.
- `new Module(`*string name*`)`
  Makes a new module that inherits from `Module.Global`.
- `new Module(`*string name*`,` *Module parent*`)`
  Makes a new module that inherits from *parent*.
- `new Module(`*string name*`,` *Module parent*`,` *params string[] sourceFiles*`)`
  Makes a new module that inherits from *parent*, and loads the specified files into it.
- *module*`.AddDefinitions(`*params string[] definitions*`)`
  Parses and runs each of the declarations, storing them in the tasks within *module*.
- *module*`.LoadDefinitions(`*string path*`)`
  Parses and runs each of the declarations in the specified file, storing them in the tasks within *module*.
- *module*`.LoadDirectory(`*string path*`)`
  Parses and runs each of the declarations in each of the .step and .csv files in the specified directory, storing them in the tasks within *module*.
- *module*`[`*string name*`]`
  Gets or sets the initial value of the global variable *name*.
- *module*`.ParseAndExecute(`*string body*`)`
  Parses and runs the specified body and returns the text generated by it.
- *module*`.Call(`*string taskName, object[] args*`)`
  Calls the specified task with the specified arguments, and returns the text it generates, or null, if the call failed.
- *module*`.CallPredicate(`*string taskName, object[] args*`)`
  Calls the specified task with the specified arguments, and returns true if it succeeds, false if not. Any text output or variable bindings are discarded.
- *module*`.CallFunction<T>(`*string taskName, object[] args*`)`
  This relies on the logic programming convention that when representing functions as predicates, the "output" of the function is the last argument. Calls the specified task with the specified arguments, followed by an unbound local variable, and returns the value of that variable. Any text output or variable bindings are discarded. Throws `CallFailedException` if the call fails. Thus, using the code from the CSV file CSV files, the call:

  ```
  m.CallFunction<string>("PluralForm", "reptoid")
  ```

  would run the Step query `[PluralForm reptoid ?]`, which would bind `?` to `reptoids`. Thus, the CallFunction call above, would return the C# string `"reptoids"`.

States

By default, the system starts each call with a fresh global state in which the global variables have their default values from the module, and any other internal state such as cooldowns or the states of mutable predicates are empty. However, as the system performs mutations on the above, it tracks them in a `Step.State` object, which is effectively a dictionary mapping state elements to their current values. `States` are opaque objects, but you can save them across calls to allow your code to maintain global state.

You can use

- `State.Empty`
  The initial state with no information in it. Any global variables have their initial values as declared in the module.
- (*string*, *state*) *module*`.Call(`*State s, string taskName, object[] args*`)`
  Calls the specified task with the specified arguments, and the specified dynamic state, and returns the text it generates, or null, if the call failed, as well as the resulting new dynamic state.
- *module*`.CallPredicate(`*State s, string taskName, object[] args*`)`
  Calls the specified task with the specified arguments and global state. Returns true if it succeeds, false if not. Any text output or variable bindings are discarded.
- *module*`.CallFunction<T>(`*State s, string taskName, object[] args*`)`
  Calls the specified function (see above) with the specified arguments and state, and returns its value for the arguments.

Defining new primitive tasks

Primitive tasks are represented by instances of the class `Step.Interpreter.PrimitiveTask`. You can add your own primitive tasks that access information in your game by creating new instances and storing them inside your module.

These objects are basically wrappers for C# methods that do the actual work. Their constructors generally take one or more functions that the interpreter can call with the arguments to the task and that return information about its success and output. These wrapped methods are referred to as the "implementation" of the task and are most often specified with lambda expressions.

Because *Step* allows tasks whose arguments might be inputs in one call and outputs in another (depending on whether the argument was passed an unbound variable), some of these wrappers require multiple such functions to handle the different cases of which arguments are in "input mode" and which are in "output mode."

Use of IEnumerable types in primitives

Tasks that can succeed more than once are effectively coroutines: they generate values one at a time and store their state so they can generate the next value when asked. In .NET, and therefore C#, these generators are represented with the `IEnumerable` and `IEnumerable<T>` interfaces, which provide an interface for asking for elements of a collection one at a time. The classes that implement such primitive tasks therefore want implementation functions that return `IEnumerables`. Those IEnumerables can be explicit coroutines (methods that return `IEnumerable` and contain the `yield return` statement), or they can be normal methods that return an `IEnumerable` object such as an array.

Classes for creating primitive tasks

You can create a new primitive task by calling one of the following constructors. Then just store the result in your module, for example:

```
myModule["Odd"] = new SimplePredicate<int>("Odd", n => (n&1) != 0);
```

will add a new task that tests whether integers are odd numbers. The wrapper will take care of checking argument types and count.

Here are the most common class constructors for primitive tasks:

- ```
  new DeterministicTextGenerator(string name,
                                 Func<IEnumerable<string>> implementation)
  new DeterministicTextGenerator<T1>(string name,
                                     Func<T1, IEnumerable<string>> implementation)
  new DeterministicTextGenerator<T1, T2>(string name,
                                         Func<T1, T2, IEnumerable<string>> impl)
  ```
  Creates a new primitive task with the specified *name*. The task takes arguments of types T1, T2 if specified. The implementation outputs a set of strings all of which are printed as separate tokens to the output, and succeeds once (i.e. can't be backtracked).

- ```
  new NonDeterministicTextGenerator(string name,
                                    Func<IEnumerable<string>> implementation)
  new NonDeterministicTextGenerator<T1>(string name,
                                        Func<T1, IEnumerable<string>> impl)
  new NonDeterministicTextGenerator<T1, T2>(string name,
                                            Func<T1, T2, IEnumerable<string>> impl)
  ```

  Creates a new primitive task with the specified *name*. The task takes arguments of types T1, T2 if specified. Use this if you want the task to be backtrackable, that is, to be able to generate one output, and then generate a different output if it is backtracked. Unlike DeterministicTextGenerator, which outputs all the enumerated strings at once, this outputs them one at a time, switching the output each time it's backtracked.

- ```
  new GeneralPredicate<T1>(string name,
                           Func<T1, bool> inMode,
                           Func<IEnumerable<T1>> outMode)
  ```
  Creates a new primitive predicate of one argument with the specified *name*. When the argument is a constant, it calls inMode and succeeds iff it returns false. If the argument is an unbound variable, it calls outMode. If outMode generates a result, it unifies the output with the first result. Should it backtrack, it unifies its output with the next element generated by outMode and succeeds again. If it runs out of elements generated by outMode, it fails. The predicate prints no output.

- ```
  new GeneralPredicate<T1, T2>(string name, Func<T1, T2, bool> inInMode,
                               Func<T1, IEnumerable<T2>> inOutMode,
                               Func<T2, IEnumerable<T1>> outInMode,
                               Func<IEnumerable<(T1, T2)>> outOutMode)
  ```
  This works the same as the previous case but for a two-argument predicate. Since there are two arguments, there are four possible input/output cases, and so four different implementation methods.

- ```
  new GeneralNAryPredicate(string name, Func<object[], IEnumerable<object[]>>
  implementation)
  ```
  ```

This is the maximally general wrapper for predicates that output no text. It calls the implementation with the raw argument array. The implementation should then enumerate a series of argument arrays each time it wants to succeed. The output array will be unified by the interpreter with the input argument array, giving you a mechanism for binding whatever variables you want to whatever values you want.

- `new SimpleFunction<TIn, TOut>(string name, Func<TIn, TOut> implementation)`
  `new SimpleFunction<TIn1, TIn2, …, TOut>(string name, Func<TIn1, TIn2, …, TOut> implementation)`
  Creates a new primitive task called *name* that takes as many arguments as there are *TIn* types plus one. These arguments must be instantiated (not be unbound variables). It calls *implementation* on those arguments and unifies its last argument with the result. It succeeds if they can be unified, otherwise fails.

- `new SimplePredicate<TIn1, TIn2, …>(string name, Func<TIn1, TIn2, …, bool> implementation)`
  Creates a new primitive task called *name* that takes as many arguments as there are *TIn* types. These arguments must be instantiated (not be unbound variables). It calls *implementation* on its arguments. If *implementation* returns true, it succeeds, otherwise it fails.

Note that there are still more general wrappers that can be used, e.g. to write higher-order primitive tasks. However, these expose more of the internal structure of the interpreter and so are less stable. If you use them, you may need to modify your code should those internal interfaces change in the future. Documentation can be found in the *Step* sources.