ACT-R RPC Interface Documentation

Working Draft

Dan Bothell

Introduction

This document contains information about a new feature available with the ACT-R 7.6+ software. There is now a built-in RPC (remote procedure call) server with the ACT-R software which makes it possible to interact with ACT-R other than through the Lisp prompt and which may make it easier to interface ACT-R to external tasks/environments. The eventual goal is to provide access to all of the ACT-R functionality through that interface, but currently only a subset of the ACT-R commands are available remotely.

Current Status

At this time all of the functionality available through the ACT-R Environment is handled through this new interface, and almost all of the tools and functions used in the ACT-R tutorial are also available (the exceptions being macros like no-output which won't work through an RPC type interface and have to be handled locally on a remote connection). There are very simple examples of connecting to the remote interface in several languages (Lisp, Python, R, C, Java, Tcl/Tk, MATLAB, and Node.js) available in the examples/connections directory of the ACT-R distribution. There are also a few more robust examples written in Python available. In the examples/creating-modules/external directory there are two different implementations of replacing the ACT-R goal module with a functionally equivalent version in Python. There is also an implementation of the protocol in Python along with functions for accessing many of the available ACT-R commands which is included with the ACT-R tutorial. That interface has been used to implement all of the tasks and experiments for the tutorial. The tutorial texts have been updated to reflect that, and it is now possible for one to work through the tutorial entirely from an interactive Python session.

General Design

The remote interface is implemented through a central command server (referred to as the dispatcher). The core ACT-R software connects to that dispatcher to provide access to its functions, and the dispatcher accepts TCP/IP socket connections which allow "clients" to access those functions as well as make their own functions available for use. Functions available through the dispatcher can be used anywhere a Lisp function was required previously (module interface functions, parameter hook functions, !eval!, etc.).

In addition to providing a means for clients to interact directly through function calls it also provides a "monitoring" capability to allow clients to be notified whenever a dispatcher function is being called regardless of who is making that call. This capability allows the dispatcher to handle some of the operations that previous versions of ACT-R implemented via hook-function parameters or other means like the methods for implementing a device. One example of how that monitoring can be used is that any client can receive the ACT-R trace and warning output by monitoring the commands which output that information (model-trace, command-trace, general-

trace, and warning-trace). Another example of the use of monitoring is that the interface for the perceptual and motor modules is being simplified to remove the need for an all-encompassing device which requires one to write specific methods for handling all of its capabilities. Instead, the perceptual and motor modules will now interact with simpler devices that are implemented for particular modalities (like a keyboard or mouse for motor actions and a window for vision) which handle the specific components necessary and provide functions which can be monitored. Thus, instead of having to write a device-handle-keypress method (or an rpm-key-down-event-handler if using the prior AGI devices) to be notified when a key is pressed, if a keyboard device is installed all a client needs to do is monitor the output-key action which it generates to detect when a key is pressed. Any number of clients may be monitoring for that action to occur, and it doesn't require knowing anything about the details of the device itself other than the action(s) it generates.

Dispatcher Details

Overview

The purpose of the dispatcher is as a central server through which any number of clients can communicate via functions which they have made available. The dispatcher keeps track of those available functions and handles the communication between the clients when they call one of the functions.

Low Level Protocol

The dispatcher opens a TCP/IP socket on port 2650 (by default if available) to which any number of clients can connect. The communication protocol is a subset of JSON-RPC version 1.0 (see appendix for notes on why this was chosen and the unused feature) with the addition of a terminating character (ASCII code 4) after every JSON object so that one doesn't have to parse the incoming stream incrementally to find the end of a message. Thus, although the dispatcher is being referred to as a server, at the ACT-R protocol level each of the "clients" is effectively making a peer-to-peer connection since all connections are able to initiate calls to functions available from any connection.

When the dispatcher starts it writes the IP address and port number it is using to files named act-r-address.txt and act-r-port-num.txt respectively in the current user's home directory (determined using ~ e.g. "~/act-r-address.txt"). That can be useful for external systems to know how to connect to ACT-R (it is used by the Python interface included with the tutorial and by the ACT-R Environment). It also writes a specific configuration file for the included ACT-R Environment in the "ACT-R:environment;GUI;init;05-current-net.tcl" as a safety measure in the event that the users home directory is not accessible. By default the dispatcher will only allow connections coming from the same machine on which it is running, but if external connections are needed the variable *allow-external-connections* can be set to a non-nil value to let the dispatcher accept them.

Note on the JSON encoding

Because Lisp uses the same representation, **nil**, for false and the empty list the dispatcher will use the JSON null value to represent both of those from calls to functions implemented in Lisp i.e. it does not use the JSON false value. Similarly, the dispatcher makes no distinction between the JSON false and null values which both get mapped onto **nil** in Lisp. Thus, from the dispatcher's perspective either null or false is a "false" value when receiving messages through the TCP/IP connections.

When returning an error result the dispatcher uses a JSON object with a single element named message whose value is the string indicating the error reason, and client connections should use the same approach for indicating error results.

In Lisp, ACT-R uses symbols for the names of things (chunks, chunk-types, productions, etc.). Because JSON does not have a symbol type, symbols are translated to strings when being sent from ACT-R, and of course Lisp strings are also transmitted as strings. In most places the distinction between whether a string represents a name or not is well specified by the context, but there are some places where either a name or string could be specified and that difference would matter (the value of a chunk's slot being one significant instance). In most of those situations a name (symbol) is the item used more often. Therefore when an ACT-R command could use a name or a string the approach will be to assume that a string represents a name, and if a string is needed then it must be specified using an additional set of single quotes in the string – if the first and last characters of a string are the single quote character then that item will be treated as a string of the characters between those single quotes i.e. "'string'" will be treated as the string "string". When a command requires the additional string encoding it will be noted in the reference manual and also indicated in the documentation string of the command (see the appendix for how that will be noted).

Many of the ACT-R commands use keyword parameters in Lisp, but there is no support for named parameters like that in the chosen RPC protocol. To allow the same sort of configurability for remote commands keyword parameters will be approximated by a construct we will call an options list. An options list can be used to provide a set of named parameters to an ACT-R command where the ordering of those parameters does not matter. There are two ways that can be provided. The straight forward approach for creating one is to provide a list of two element lists where each sub-list consists of the parameter name and the parameter value respectively. The other approach involves specifying a JSON object to send to the command where the members of that object represent the parameters and values. The reason for allowing two representations is because one or the other may be easier to use in a remote language e.g. a dictionary in Python is encoded into JSON as an object with the keys and values as the members of that object thus a Python dictionary could be passed as an options list to an ACT-R command.

High Level Protocol

The operation of the system is handled using a fixed set of commands which are accepted and generated by the dispatcher -- it is not a direct RPC interface between clients. The RPC interface is used to provide the commands that implement the ACT-R remote interface and the dispatcher acts as the intermediary between all the connected components. There are nine methods (the JSON-RPC terminology) which the dispatcher will accept (which fall into four categories) and the clients must support one method to handle calls made to it if it adds any commands.

In the descriptions of the methods below the return result is only valid if it completes successfully. If it doesn't then the error result will include a string indicating what went wrong.

Adding/removing dispatcher commands

Method: add

Parameters: published-name, private-name, {documentation}, {single-instance?}, {Lisp-cmd?}

Returns: published-name

Description:

The add method is used to add a new command to those available from the dispatcher. It requires two strings. The first indicates the name which can be used to evaluate this action from the dispatcher and is case sensitive i.e. "dm" is different from both "DM" and "Dm". The second is the name which the dispatcher will provide to the client when requesting the evaluation and is also case sensitive. They can be the same name but do not have to be (one reason that might be useful is if one wants multiple external commands to all be handled by the same method in the client's interface). The second parameter may also be the JSON value null or false, which means that there is no underlying command for evaluation. That can be useful when one wants to provide a command that can be monitored, but which has no underlying functionality (those are referred to as signals in the ACT-R docs). An optional documentation string can be provided as the third parameter. The recommendation is that the documentation string should always be given and at least provide the parameters to be used with the command (see the appendix for the syntax that is used for indicating details of the parameters in the ACT-R commands' documentation strings). The optional fourth parameter can be used to indicate whether the dispatcher should protect against multiple concurrent evaluations of this command. If it is a string then that string will be returned as the error result if an evaluation of this published command is attempted while one is still pending, and if it is any other non-false value then it will suspend evaluation of subsequent attempts until prior attempts have been completed. If the optional fifth parameter is provided it must be a string which will be used to create a macro in Lisp for accessing this command. That string will be upcased and used to name a macro which will quote all of the parameters provided to it and pass them to the command and return

the result. [Note: if you use Allegro Common Lisp with the IDE a macro created by using this command might not be directly useable in the Debug Window of the IDE because of a bug in their IDE not being able to find its argument list to display in the status line, but it does work properly everywhere else, particularly in model definitions which is its intended purpose.]

When successful it returns the published name that was provided.

Method: remove

Parameters: published-name

Returns: published-name

Description:

The remove method is used to remove a command which has been added to the dispatcher. It requires a single parameter which is the published name of that command. That command will be removed from those which are available. Currently, there is no protection on the commands and any client may remove any command – not just its own.

When successful it returns the published name that was provided.

Monitoring commands

Method: monitor

Parameters: command-to-monitor, command-to-call, {monitor-style}

Returns: command-to-call

Description:

The monitor method is used to have a command evaluated automatically when another command is evaluated. The command-to-monitor (also referred to as the monitored command) should be a string indicating the command which is being monitored, and command-to-call is a string indicating the command to call automatically (also referred to as a monitoring command). When the monitored command is evaluated the monitoring command will also be evaluated (note that the automatic evaluation of the monitoring command does not trigger any commands which are monitoring it – only explicit evaluations will trigger a monitoring command to be evaluated). The optional monitor-style parameter should not be used at this time (it was originally designed to support different ways for the automatic evaluation to occur, but that functionality is not documented at will likely be removed in the near future). The evaluation of the monitoring command(s) will be made after the monitored command has returned, and the monitoring

command(s) will be passed the same list of parameters as the monitored command was passed. The return values from any monitoring commands are ignored.

When successful it returns the name of the monitoring command.

Method: remove-monitor

Parameters: command-to-monitor, command-to-call

Returns: command-to-call

Description:

The remove-monitor method is used to remove a command monitor which has been created using the monitor method. It requires two parameters which are strings that name the monitored and monitoring commands which were given in a previous use of the monitor method. Removing the monitor stops the automatic evaluations of the monitoring command when the monitored command is evaluated.

When successful it returns the name of the command which was previously monitoring.

Evaluating dispatcher commands

Method: evaluate

Parameters: command, model, {parameters}*

Returns: list of results

Description:

The evaluate method is used to evaluate a command from the dispatcher. It requires two parameters. The first is a string with the name of the command to evaluate, and the second is either a string containing the name of a model in which the evaluation should be performed or a false value indicating that no specific model is required (typically that means the current model will be used but is up to the implementer of the command being evaluated to actually decide how to handle the model value provided). The remainder of the parameters to the evaluate method are those which will be passed to the command to be evaluated, in the order given.

When successful it returns the list of results from evaluating that command.

Dispatcher information

Method: check

Parameters: command-to-check

Returns: command status

Description:

The check method is used to determine if a command has been added to the dispatcher. It requires one parameter which is a string with the name of the command to check. If that string names a command which has been added to the server then three values are returned. The first will be true. The second will be true if the client that performed the check was the one to add it otherwise it will be null, and the third will be the documentation string of that command. If that string does not name a command which has been added to the server then a single value of null will be returned.

Method: list-commands

Parameters:

Returns: list of results

Description:

The list-commands method can be used to get a list of all the currently available commands from the dispatcher. It requires no parameters and returns a list of two element lists. Each sub-list contains two strings. The first is the name of a command available through the dispatcher and the second is the documentation string which was provided when the command was added.

Method: set-name

Parameters: name-string

Returns: name-string

Description:

The set-name method is used to indicate a name for the connected client when inspecting the current connections. It requires one parameter which is a string with the name for the client. If a string is provided then that will be used as the client's name and that string will be returned. If the name is not a string null will be returned. Note that names do not need to be unique – multiple clients may report the same name. It is also possible for a client to change its name because there are no restrictions on when or how often a set-name method may be used by a client.

Method: list-connections

Parameters:

Returns: list of connection lists

Description:

The list-connections method can be used to get a list of all the clients currently connected to the dispatcher (including the internal connection). It requires no parameters and returns a list lists, with each sub-list containing 5 items describing a connected client. The first item will be the string with the client's name if set otherwise null. The second will be a string with the ip-address of the client. The third will be a count of the number of method requests which have been received from the client and not yet completed. The fourth will be a count of the number of evaluation requests which have been sent to the client and not yet returned. The fifth will be a list of the names of commands which the client has added (in alphabetical order). Note that the internal connection does not track method requests and evaluations thus those values will be reported as -1, and for the ip-address, the internal connection returns the address to which clients can connect.

Client Side Evaluation

Method: evaluate

Parameters: command, model, {parameters}*

Returns: list of results

Description:

The only method which will be requested of clients is evaluate, and that will only happen if they have added any commands. The client side evaluate method is just like the server side one, except that the command name sent is the private name used to evaluate a command from the dispatcher. It will always be passed at least two parameters. The first is a string with the private name of the command to evaluate, and the second is either a string containing the name of a model in which the evaluation should be performed or a false value indicating that no specific model is required. The remainder of the parameters to the evaluate method are those which are being passed to the command to be evaluated, in the order given.

A list of the return values from the command should be returned if it completes successfully. If it does not complete successfully then the error result should contain an object with a single slot named message which contains a string describing the error which occurred.

Lisp Dispatcher Commands

The following commands are available in the Lisp which is running the dispatcher. Effectively, the client interface is available from Lisp without the need of the socket communication and the JSON encoding, and there are also commands which allow the evaluation of either Lisp functions or dispatcher commands named with strings.

add-act-r-command

Syntax:

add-act-r-command *name* {*function*} {*documentation*} {*single-instance?*} {*macro*} -> [**t** | **nil**], details

Arguments and Values:

name ::= a string containing the name of the command to add
function ::= [**nil** | a symbol naming a function]
documentation ::= [**nil** | a string containing the documentation for the command]
single-instance? ::= [**t** | **nil** | a string containing an error message]
macro ::= [**nil** | a string containing the name of a macro to create]
details ::= [name | a string containing the failure details]

Description:

This is the internal version of the add method for the dispatcher. It adds a command to those available through the dispatcher with the given name that will be implemented by the specified function (if provided). If single-instance? is provided as **t** (the default) then only one instance of the command function will be allowed to execute at any time and subsequent attempts will be blocked until the currently executing one completes. If single-instance? is a string then a request to evaluate the command while one is running will not succeed and an error result with the string provided will be returned. If single-instance? is **nil** then there is no limit to how many of the functions can be evaluated simultaneously. If macro is provided then a macro with that name is created which will pass all the parameters provided to it to this command through the dispatcher and return the result.

If a command is successfully added to the dispatcher then **t** is returned as the first value and the name of the command as the second. If the command cannot be added then **nil** is returned and the second value is a string indicating the reason it could not be added.

Examples:

```
> (add-act-r-command "print-warning" 'print-warning-internal
     "Send a string to the ACT-R warning-trace. Params: <warning-string>." nil)
```

```
T
"print-warning"

E> (add-act-r-command 'bad-name)
NIL
"Invalid parameters when trying to add command specified with name BAD-NAME and
function NIL"
```

remove-act-r-command

Syntax:

remove-act-r-command *name* -> [**t** | **nil**], details

Arguments and Values:

name ::= a string containing the name of the command to add
details ::= [name | a string containing the failure details]

Description:

This is the internal version of the remove method for the dispatcher. It removes the named command from those that are available through the dispatcher.

If a command is successfully removed then **t** is returned as the first value and the name of the command as the second. If the command cannot be removed then **nil** is returned and the second value is a string indicating the reason it could not be removed.

Examples:

```
> (remove-act-r-command "goal-focus")
T
"goal-focus"

E> (remove-act-r-command 'bad-name)
NIL
"Error #<SIMPLE-ERROR Invalid remove name> occurred while trying to remove command
specified by (BAD-NAME)"
```

evaluate-act-r-command / call-act-r-command / dispatch-apply

Syntax:

evaluate-act-r-command *name* {*params*}* -> [**t** | **nil**], results*
call-act-r-command *name* {*params*}* -> [**nil |** results*]
dispatch-apply *fname* {*params*}* -> [**nil |** results*]

Arguments and Values:

name ::= a string containing the name of a dispatcher command

params ::= any value to pass to the command
results ::= the values returned by the command
fname ::= [name | a valid function designator]

Description:

These commands provide a way to evaluate dispatcher commands. The differences between them are in how the return results are provided and what commands are valid. Evaluate-act-r-command returns a value of **t** if it successfully evaluates the command followed by any values returned from the evaluation, and if the evaluation fails it returns **nil** followed by a string indicating why it failed. Call-act-r-command returns the values of a successful evaluation of the command, but if the evaluation fails it returns **nil** and an ACT-R warning is printed with the reason for the failure. Dispatch-apply is similar to call-act-r-command except that the name provided can be a string naming a dispatcher command or a Lisp function, and when the name is a Lisp function it will just be applied to the parameters without going through the dispatcher (with no additional error protection).

Examples:

```
> (evaluate-act-r-command "pprint-chunks" 'free)
FREE
   NAME  FREE

T
(FREE)

> (call-act-r-command "pprint-chunks" 'free)
FREE
   NAME  FREE

(FREE)

> (dispatch-apply "pprint-chunks" 'free)
FREE
   NAME  FREE

(FREE)

> (dispatch-apply 'pprint-chunks-fct '(free))
FREE
   NAME  FREE

(FREE)

E> (evaluate-act-r-command "bad-name")
NIL
"Command \"bad-name\" not found in the table and cannot be executed."

E> (call-act-r-command "bad-name")
#|Warning: Error "Command \"bad-name\" not found in the table and cannot be executed."
while attempting to call the ACT-R command ("bad-name" ) |#
NIL

E> (dispatch-apply "bad-name")
#|Warning: Error "Command \"bad-name\" not found in the table and cannot be executed."
while attempting to evaluate the form ("bad-name" ) |#
```

```
NIL

E> (dispatch-apply 'bad-name)
#|Warning: Function BAD-NAME provide for dispatch-apply is not a local or dispatcher
function |#
NIL
```

monitor-act-r-command

Syntax:

monitor-act-r-command *command-to-monitor command-to-call* {*when*} -> [**t** | **nil**], result

Arguments and Values:

command-to-monitor ::= a string containing the name of a dispatcher command
command-to-call ::= a string containing the name of a dispatcher command
when ::= any value (Note: should not be used)
result ::= [command-to-call | a string containing a failure reason]

Description:

This command is equivalent to the monitor method of the dispatcher. The command-to-monitor should be a string indicating the command which is being monitored, and command-to-call is a string indicating the command to call automatically. When the monitored command is evaluated the monitoring command will also be evaluated (note that the automatic evaluation of the monitoring command does not trigger any commands which are monitoring it – only explicit evaluations will trigger a monitoring command to be evaluated). The optional when parameter should not be used (it is not documented and will likely be removed in the near future). The evaluation of the monitoring command will occur after the monitored command has returned, and the monitoring command will be passed the same list of parameters as the monitored command was passed. Any return values of the monitoring command will be ignored.

It returns **t** and the name of the command to call if the monitoring is setup successfully. If there is a problem with enabling the monitoring then **nil** is returned along with a string indicating the reason for the problem.

Examples:

```
> (monitor-act-r-command "print-warning" "model-output")
T
"model-output"

E> (monitor-act-r-command "bad-name" "model-output")
NIL
"Command \"bad-name\" does not exist so command \"model-output\" cannot be called
after it."
```

remove-act-r-command-monitor

Syntax:

remove-act-r-command-monitor *command-to-monitor command-to-call* -> [**t** | **nil**], result

Arguments and Values:

command-to-monitor ::= a string containing the name of a dispatcher command
command-to-call ::= a string containing the name of a dispatcher command
result ::= [command-to-call | a string containing a failure reason]

Description:

This command is equivalent to the remove-monitor method of the dispatcher. The command-to-monitor should be a string indicating a command which is being monitored, and command-to-call is a string indicating the command which is monitoring it as setup by monitor-act-r-command (or the monitor method of the dispatcher). This command will remove that monitoring.

It returns **t** and the name of the command to call if the monitoring removal is successfully. If there is a problem with removing the monitoring then **nil** is returned along with a string indicating the reason for the problem.

Examples:

```
> (remove-act-r-command-monitor "print-warning" "model-output")
T
"model-output"

E> (remove-act-r-command-monitor "bad-name" "model-output")
NIL
"Command \"bad-name\" does not exist so monitor \"model-output\" does not need to be
removed."
```

list-act-r-commands

Syntax:

list-act-r-commands -> ((name doc)*)

Arguments and Values:

name ::= a string naming an available dispatcher command
doc ::= [**nil** | a string containing the command's documentation]

Description:

This command returns a list of two element lists where each sub-list contains the name of a command which has been added to the dispatcher and the second element will be the documentation string specified for that command or **nil** if no documentation string was given. There will be a sub-list for every command which is currently available through the dispatcher.

Examples:

```
> (list-act-r-commands)
(("sdm" "Print the chunks from declarative memory which match with the given
specification.  Params: <spec>* where spec ::= {modifier} <slot-name> <slot-value>")
("add-button-to-exp-window" "Create a button item for the provided experiment window
with the features specified and place it in the window. Params: <window> <text> {x {y
{action-command {height {width {color}}}}}}.")
 ("permute-list" "Return a randomly ordered copy of the list provded using the current
model's random stream. Params: <list>.") ...)
```

check-act-r-command

Syntax:

check-act-r-command *name* -> [**nil** | **t**, owner, doc]

Arguments and Values:

name ::= a string naming a command to check
owner ::= [**t** | **nil**]
doc ::= [**nil** | a string containing a command's documentation]

Description:

This command is used to determine whether a command with the indicated name has been added to the dispatcher. If the provided name does not name a command available from the dispatcher then **nil** is returned. If the name does name an available command then three values are returned. The first is **t**. The second will be **t** if the command was added from this Lisp session or **nil** if it was added in any other way. The third parameter will be the doc string provided when the command was added, or **nil** if it does not have a doc string.

Examples:

```
> (check-act-r-command "add-dm")
T
T
"Create chunks in the current model and add them to the model's declarative memory.
Params: {chunk-description}*."

> (check-act-r-command "Bad-name")
NIL
```

```
> (check-act-r-command "remote-cmd")
T
NIL
"No documentation provided."
```

local-or-remote-function-p / local-or-remote-function-or-nil

Syntax:

local-or-remote-function-p *name* -> [**t** | **nil**]
local-or-remote-function-or-nil *name* -> [**t** | **nil**]

Arguments and Values:

name ::= any object

Description:

Local-or-remote-function-p can be used to test whether a value is a string that names an available dispatcher command or is a valid Lisp function designator, and local-or-remote-function-or-nil will test whether the item is a string that names an available dispatcher command, a valid Lisp function designator, or **nil**. Those are useful when creating "hook function" parameters for a module which will allow Lisp or dispatcher functions to be supplied.

Examples:

```
> (local-or-remote-function-p "add-dm")
T

> (local-or-remote-function-p t)
NIL

> (local-or-remote-function-p nil)
NIL

> (local-or-remote-function-p (lambda ()))
T

> (local-or-remote-function-or-nil "add-dm")
T

> (local-or-remote-function-or-nil t)
NIL

> (local-or-remote-function-or-nil nil)
T

> (define-parameter :sim-hook :valid-test 'local-or-remote-function-or-nil
                    :default-value nil
                    :warning "a function, string naming a command, or nil"
                    :documentation "Similarity hook")
```

Appendix

Why JSON-RPC 1.0?

A peer-to-peer protocol is necessary since either side may need to call into the other (obviously the clients will need to call into the ACT-R system but in many situations the dispatcher will also need to initiate a call to the clients to evaluate any commands that they've added). Something light weight and easy to generate and decode are also useful features. Looking for an existing protocol that meets those criteria turned up JSON-RPC, version 1.0 (there is also a newer 2.0 version but it is not peer-to-peer and thus doesn't really fit with what's needed). It does have one complication which doesn't seem completely necessary for what is needed: the "class hinting" mechanism (it could be useful for some things but at this point it seems more complicated than what we need). Thus, at this point the protocol for communicating with the dispatcher does not support the class hinting mechanism. One thing that the 2.0 version protocol has which would be useful is named parameters, since ACT-R uses a lot of keyword parameters in Lisp, but for simplicity at this point the remote interface to the ACT-R commands will use the option list representation described above in place of directly named parameters.

Documentation parameter syntax

In the documentation strings for the commands provided by ACT-R it lists the parameters for using them, and the same approach is recommended for any commands which are added. In those descriptions the following syntax applies:

- parameters are required unless marked otherwise
- any parameters inside a set of single quotes means those parameters require the additional encoding to differentiate strings from names
- items in curly braces {} are optional parameters
- items in parentheses ( ) indicate a list of items
- an asterisk * after an item means 0 or more iterations can be given
- a plus sign + after an item means 1 or more iterations can be given
- angle brackets < > represent an options list, and the valid names are separated by commas