Canny Bag o' Tudor

An experimental Prolog 'pack' comprising technical spikes, or otherwise useful, Prolog predicates that do not seem to fit anywhere else

Roy Ratcliffe

SWI-Prolog

Contents

Chapter 1

Canny bag o' Tudor

!cov !fail

See PDF for details.

This is an experimental SWI-Prolog 'pack' comprising technical spikes, or otherwise useful, Prolog predicates that do not seem to fit anywhere else.

The package name reflects a mixed bag of bits and pieces. It's a phrase from the North-East corner of England, United Kingdom. 'Canny' means nice, or good. Tudor is a crisp (chip, in American) manufacturer. This pack comprises various unrelated predicates that may, or may not, be tasty; like crisps in a bag, the library sub-folders and module names delineate the disparate components. If the sub-modules grow to warrant a larger division, they can ultimately fork their own pack.

The pack comprises experimental modules subject to change and revision due to its nature. The pack's major version will always remain 0. Work in progress!

Chapter 2

library(canny/a)

a_star(+*Heuristics0*, -*Heuristics*, +*Options***)** *[det]*

Offers a static non-Constraint Handling Rules interface to `a_star/4`. Performs a simplified A* search using CHR where the input is a list of *all the possible* arcs along with their cost. Each element in *Heuristics0* is a `h/3` term specifying source of the heuristic arc, the arc's destination node and the cost of traversing in-between. Nodes specify distinct but arbitrary terms. Only terms `initial` and `final` have semantic significance. You can override these using *Options* for `initially` and `finally`. For *Options* see below.

Simplifies the CHR implementation by accepting `h/3` terms as a list rather than using predicates to expand nodes. We match heuristic terms using `member/2` from the list of heuristics. This interface does **not** replace `a_star/4` since having a pre-loaded list of heuristics is not always possible or feasible, for example when the number of arcs is very large such as when traversing a grid of arcs.

Here is a simple example.

```
?- a_star([h(a, b, 1)], A, [initially(a), finally(b)]).
A = [h(a, b, 1)].
```

Options include:

- `initially(Initial)` defines the initial node, defaults to atom `initial`.
- `finally(Final)` defines the final node, atom `final` by default.
- `reverse(Boolean)` reverses the outgoing selected *Heuristics* so that the order reflects the forward order of traverse. The underlying expansion pushes path nodes to the head of the list resulting in a final-to-initial traversal by default.

See also `https://rosettacode.org/wiki/A*_search_algorithm`

Chapter 3

library(canny/arch)

current_arch(*?Arch:pair***)** *[semidet]*

Unifies *Arch* with the current host's architecture and operating system. Usefully reads the pair as a Prolog term with which you can unify its component parts.

The Prolog `arch` flag combines both the architecture and the operating system as a dash-separated pair. The predicate splits these two components apart by reading the underlying atom as a Prolog term. This makes an assumption about the format of the `arch` flag.

current_arch_os(*?Arch, ?OS***)** *[semidet]*

Unifies *OS* with the current operating system. Splits the host architecture into its two components: the bit-wise sub-architecture and the operating system. Operating system is one of: `win32` or `win64` for Windows, `darwin` for macOS, or `linux` for Linux. Maps architecture bit-width to an atomic *Arch* token for contemporary 64-bit hosts, one of: `x64`, `x86_64`. Darwin and Linux report the latter, Windows the former.

current_os(*?OS***)** *[semidet]*

Succeeds for current *OS*, one of:

- `win32`
- `win64`
- `darwin`
- `linux`

Chapter 4

library(canny/arity)

arities(*?Arities0:compound, ?Arities:list***)** *[semidet]*

Suppose that you want to accept arity arguments of the form {A, ...} where A is the first integer element of a comma-separated list of arity numbers. The *Arities0* form is a compound term enclosed within braces, comprising integers delimited by commas. The `arities/2` predicate extracts the arities as a list.

Empty lists fail. Also, lists containing non-integers fail to unify. The implementation works forwards and backwards: arity compound to arity list or vice versa, mode (+, -) or mode (-, +).

Chapter 5

library(canny/bits)

bits(+*Shift, +Width, ?Word, ?Bits, ?Rest***)** *[semidet]*
bits(+*ShiftWidthPair, ?Word, ?Bits, ?Rest***)** *[semidet]*
bits(+*ShiftWidthPair, ?Word, ?Bits***)** *[semidet]*

Unifies *Bits* within a *Word* using *Shift* and *Width*. All arguments are integers treated as words of arbitrary bit-width.

The implementation uses relational integer arithmetic, i.e. CLP(FD). Hence allows for forward and backward transformations from *Word* to *Bits* and vice versa. Integer *Word* applies a *Shift* and bit *Width* mask to integer *Bits*. *Bits* is always a smaller integer. Decomposes the problem into shifting and masking. Treats these operations separately.

Arguments

Width	of *Bits* from *Word* after *Shift*. *Width* of zero always fails.

bit_fields(+*Fields:list, +Shift:integer, +Int:integer***)** *[semidet]*
bit_fields(+*Fields:list, +Shift:integer, +Int0:integer, -Int:integer***)** *[semidet]*

Two versions of the predicate unify *Value*:*Width* bit fields with integers. The arity-3 version requires a bound *Int* from which to find unbound (or bound) values in the *Fields*; used to extract values from integers else check values semi-deterministically. The arity-4 version of the predicate accumulates bit-field values by OR-wise merging shifted bits from *Int0* to *Int*.

The predicates are semi-deterministic. They can fail. Failure occurs when the bit-field *Width* integers do **not** sum to *Shift*.

Arguments

Fields	is a list of value and width terms of the form `Value:Width` where *Width* is an integer; *Value* is either a variable or an integer.
Shift	is an integer number of total bits, e.g. 8 for eight-bit bytes, 16 for sixteen-bit words and so on.

rbit(+*Shift:integer, +Int:integer, ?Reverse:integer***)** *[semidet]*

Bit reversal over a given span of bits. The reverse bits equal the mirror image of the original; integer 1 reversed in 16 bits becomes 8000_{16} for instance.

Arity-3 `rbit/3` predicate throws away the residual. Any bit values lying outside the shifting span disappear; they do not appear in the residual and the predicate discards them. The order of the sub-terms is not very important, except for failures. Placing `succ` first ensures that recursive shifting fails if *Shift* is not a positive integer; it triggers an exception if not actually an integer.

Chapter 6

library(canny/cover)

coverages_by_module(*:Goal, -Coverages:dict***)** *[det]*

Calls *Goal* within `show_coverage/1` while capturing the resulting lines of output; *Goal* is typically `run_tests/0` for running all loaded tests. Parses the lines for coverage statistics by module. Ignores lines that do not represent coverage, and also ignores lines that cover non-module files. Automatically matches prefix-truncated coverage paths as well as full paths.

Arguments

Coverages	is a module-keyed dictionary of sub-dictionaries carrying three keys: clauses, cov and fail.

coverage_for_modules(*:Goal, +Modules, -Module, -Coverage***)** *[nondet]*

Non-deterministically finds *Coverage* dictionaries for all *Modules*. Bypasses those modules excluded from the required list, typically the list of modules belonging to a particular pack and excluding all system and other supporting modules.

Chapter 7

library(canny/crc)

crc(+*Predefined*, -*CRC*)**** *[semidet]*

Builds a predefined *CRC* accumulator.

Arguments

Predefined	specifies a predefined *CRC* computation.
CRC	a newly-initialised *CRC* term with the correct polynomial, initial value and any necessary options such as bit reversal and inversion value.

crc_property(+*CRC*, ?*Property*)** *[semidet]*

Extracts the *CRC*'s checksum for comparison, or unifies with other interesting values belonging to a *CRC* accumulator.

crc(+*CRC0*, +*Term*, -*CRC*)** *[semidet]*

Mutates *CRC0* to *CRC* by feeding in a byte code, or a list of byte codes.

Arguments

CRC0	the initial or thus-far accumulated *CRC*.
Term	a byte code or a list of byte codes.
CRC	the updated *CRC*.

crc_16_mcrf4xx(-*Check*)** *[det]*

Initialises CRC-16/MCRF4XX checksum.

crc_16_mcrf4xx(+*Check0*, +*Data*, -*Check*)** *[det]*

Accumulates CRC-16/MCRF4XX checksum using optimal shifting and exclusive-OR operations.

Chapter 8

library(canny/docker): Canny Docker

author Roy Ratcliffe
version 0.1.0

This module provides an interface to the Docker API, allowing interaction with Docker services through HTTP requests. It defines settings for the Docker daemon URL and API version, and provides a predicate to construct URLs and options for various Docker operations.

It supports operations such as listing containers, creating containers, and checking the Docker system status. The module uses Prolog dictionaries to represent JSON data structures, making it easy to work with the Docker API's responses. It also includes utility predicates for transforming dictionary key-value pairs and constructing paths for API requests. It is designed to be used in conjunction with the HTTP client library to make requests to the Docker API. It provides a flexible way to interact with Docker services, allowing for dynamic construction of API requests based on the specified operations and options.

8.1 Docker API Operations

The module supports various Docker API operations, such as:

- `system_ping`: Check if the Docker daemon is reachable.
- `container_list`: List all containers.
- `container_create`: Create a new container.
- `network_create`: Create a new network.
- `network_delete`: Delete a network.

These operations are defined in the Docker API specification and can be accessed through the `docker/3` predicate, which constructs the appropriate URL and options based on the operation and the settings defined in this module.

8.1.1 Example container operations

The following examples demonstrate how to list and create Docker containers using the `docker/3` predicate. The first example lists all containers, and the second example creates a new container with a specified image and labels.

```
?- docker(container_list, Reply).
Reply = [json(['Id'='abc123', 'Image'='ubuntu:latest', ...|...])].
?- docker(container_create, Reply, [post(json(json(['Image'=ubuntu,
   'Labels'=json(['Hello'=world)])))])]).
Reply = _{Id:"abc123", Warnings:[]}.
```

The `container_list/2` predicate retrieves a list of all containers, returning a list of dictionaries representing each container. Each dictionary contains information such as the container ID, image, and other metadata. The `container_create/3` predicate creates a new container with the specified image and labels. The labels are specified as a JSON object, allowing for flexible tagging of containers with metadata. The reply contains the ID of the created container and any warnings that may have occurred during the creation process. The labels can be used to organise and manage containers based on specific criteria, such as purpose or owner.

8.1.2 Example network operations

The following examples demonstrate how to create and delete a Docker network using the `docker/3` predicate. The network is created with a name and labels, and then deleted by its name.

```
?- docker(network_create(_{name:my_network, labels:_{'my.label':'my-value'}}), A).
A = _{id:"1be0f5d2337ff6a6db79a59707049c199268591f49e3c9054fc698fe7916f9c3", warning:""}.

38 ?- docker(network_delete(my_network), A).
A = ''.
```

Note that the `network_create/2` predicate constructs a network with the specified name and labels, and returns a reply containing the network ID and any warnings. The `network_delete/2` predicate deletes the network by its name, returning an empty reply if successful.

Labels can be used to tag networks with metadata, which can be useful for organising and managing Docker resources. The labels are specified as a dictionary with key-value pairs, where the keys and values are strings. The labels are included in the network configuration when creating a network, allowing for flexible and dynamic tagging of Docker resources.

Labels can be used to filter and query networks, making it easier to manage Docker resources based on specific criteria. For example, you can create a network with a label indicating its purpose or owner, and then use that label to find networks that match certain criteria. This allows for more organised and efficient management of Docker resources, especially in larger deployments with many networks and containers.

8.1.3 Restyling Keys

The `docker/3` predicate transforms the keys in the input dictionary to CamelCase format using the `restyle_key/3` predicate, which applies the Docker-specific CamelCase naming convention to the keys. This transformation is useful for ensuring that the keys in the input dictionary match the expected format for the Docker API, making it easier to work with the API and ensuring compatibility with the expected request format.

The transformation is applied recursively to all key-value pairs in the input dictionary, ensuring that all keys are transformed to the correct format before making the request to the Docker API. The reverse transformation is applied to the reply dictionary, which does not retain the original key names as returned by the Docker API. Label keys are also transformed to CamelCase format, ensuring consistency in the naming convention used for labels in the Docker API requests and responses.

8.2 Low-Level HTTP Requests

The module provides a low-level interface to the Docker API, allowing for custom HTTP requests to be made. The `docker/3` predicate constructs the URL and options for the specified operation, and uses the `http_get/3` predicate to make the request. The options can include HTTP methods, headers, and other parameters as needed for the specific operation.

The `url_options/4` predicate is used to construct the URL and options for a specific Docker operation. It retrieves the operation details from the Docker API specification and formats the path according to the specified version and operation. The resulting URL and options can be used with the HTTP client to make requests to the Docker API.

8.2.1 Example usage

The `url_options/4` predicate can be used to construct the URL and options for a specific Docker operation. For example, to get the URL and options for the `system_ping` operation, you can use:

```
?- [library(http/http_client)].
true.

?- canny_docker:url_options(system_ping, URL, Options),
   http_get(URL, Reply, Options).
URL = [path('/v1.49/_ping'), protocol(tcp), host(localhost), port(2375)],
Options = [method(get), accept(["text/plain"])],
Reply = 'OK'.
```

For listing containers, you can use:

```
?- canny_docker:url_options(container_list, URL, Options),
   http_get(URL, Reply, Options).
URL = [path('/v1.49/containers/json'), protocol(tcp), host(localhost), port(2375)],
Options = [method(get), accept(["application/json"])],
Reply = [json(['Id'=..., ...|...])].
```

For creating a container, you can use:

```
?- docker(container_create, A, [post(json(json(['Image'=ubuntu,
   'Labels'=json(['Hello'=world])]))))]).
```

This example creates a new Docker container with the specified image and labels. Notice that the post request uses `json(json(...))` to specify the JSON body of the request.

docker(+*Ask*, -*Reply*) *[det]*

Issues a request to the Docker API using the specified *Ask* term and returns the *Reply*. The *Ask* term may be a compound specifying the operation to perform together with any required arguments.

The Docker API request comprises:

- a path with zero or more placeholders,
- a method,
- zero or more required or optional search parameters,
- a JSON body for POST requests.

This implies that, for the least amount of additional information, a request is just a path with a method, e.g., a GET, HEAD or DELETE request. From that point onward, requests grow in complexity involving or more of the following: path placeholders, query parameters, a request body.

The complexity of the request can vary significantly based on the operation being performed and the specific requirements of the Docker API. The `docker/2` predicate is designed to handle these variations and provide a consistent interface for interacting with the Docker API. It abstracts away the details of constructing the request and processing the response, allowing users to focus on the high-level operation they want to perform. Path placeholders appear in the first *Ask* term argument as atomic values. URL query parameters are specified as a list of key-value pairs in the *Ask* term argument. POST request payloads are specified as a Prolog dictionary as the *Ask* term.

The *Ask* term is a compound term that specifies the operation to perform, such as `container_list` or `system_ping`. The *Reply* is a Prolog term that represents the response from the Docker API, which is typically a Prolog dictionary or list, depending on the operation.

The predicate constructs the URL and options based on the operation and the settings defined in this module. It uses the `ask/4` predicate to determine the path, method, and any additional options required for the request. The URL is constructed by appending the path to the `daemon_url` setting, and the HTTP request is made using the `http_get/3` predicate from the HTTP client library.

The *Reply* is then processed to ensure that the keys in the response are transformed to CamelCase format using the `restyle_value/3` predicate. This transformation is useful for ensuring that the keys in the response match the expected format for the Docker API, making it easier to work with the API and ensuring compatibility with the expected response format.

Arguments

Ask	The *Ask* term specifies the operation to perform, which may include path placeholders, query parameters, and a request body. The *Ask* term is a compound term that identifies the operation and provides any necessary arguments or parameters for the request. The *Ask* term can be a simple atom for operations with no arguments, or it can be a more complex term that includes arguments. The *Ask* term is used to construct the URL and options for the request, allowing for flexible and dynamic construction of API requests based on the specified operation and options.
Reply	The *Reply* is the response from the Docker API, which is typically a Prolog dictionary or list, depending on the operation. It can also be an atom. The *Reply* is a Prolog term that represents the data returned by the Docker API after processing the request. It contains the results of the operation, such as a list of containers, the status of a container, or the result of a command.

docker(+*Operation*, -*Reply*, +*Options***)** *[det]*

Makes a request to the Docker API using the specified operation and options. The operation is a string that identifies the Docker API operation to perform, such as `container_list` or `system_ping`. The predicate constructs the URL and options based on the operation and the settings defined in this module.

Builds HTTP request options for the Docker API using the base URL from the `daemon_url` setting. The path and HTTP method are determined by `path_and_method/4`, and the resulting options are suitable for making requests to the Docker API.

The predicate constructs the URL by concatenating the base URL with the path and method. The `daemon_url` setting provides the base URL, and the `api_version` setting specifies the version of the Docker API.

Arguments

Operation	The operation to perform, which determines the path and method, as well as any additional options.
Reply	The response from the Docker API, which is typically a Prolog dictionary or list, depending on the operation.
Options	This is a list of options that control both how the path is formatted and how the HTTP request is made. For path formatting, options are terms like `id(Value)` that provide values for placeholders in the path template. For the HTTP request, options can include settings such as headers, authentication, or other parameters supported by the HTTP client.

docker_path_options(*?Operation, -Path, -Options***)** *[semidet]*

Constructs the *Path* and *Options* for a Docker API operation. The predicate retrieves the operation details from the Docker API specification and formats the path according to the default version and operation. The resulting path and options can be used with the HTTP client to make requests to the Docker API.

The predicate uses the `docker_path_options/4` predicate to construct the path and options for the specified operation. It retrieves the operation details from the Docker API specification and formats the path according to the specified version and operation. The resulting path and options can be used with the HTTP client to make requests to the Docker API.

build_prune	'/v1.49/build/prune'	post
config_create	'/v1.49/configs/create'	post
config_delete	'/v1.49/configs/{id}'	delete
config_inspect	'/v1.49/configs/{id}'	get
config_list	'/v1.49/configs'	get
config_update	'/v1.49/configs/{id}/update'	post

For container operations, the following paths and options are defined:

container_archive	'/v1.49/containers/{id}/archive'	get
container_archive_info	'/v1.49/containers/{id}/archive'	head
container_attach	'/v1.49/containers/{id}/attach'	post
container_attach_websocket	'/v1.49/containers/{id}/attach/ws'	get
container_changes	'/v1.49/containers/{id}/changes'	get
container_create	'/v1.49/containers/create'	post
container_delete	'/v1.49/containers/{id}'	delete
container_exec	'/v1.49/containers/{id}/exec'	post
container_export	'/v1.49/containers/{id}/export'	get
container_inspect	'/v1.49/containers/{id}/json'	get
container_kill	'/v1.49/containers/{id}/kill'	post
container_list	'/v1.49/containers/json'	get
container_logs	'/v1.49/containers/{id}/logs'	get
container_pause	'/v1.49/containers/{id}/pause'	post
container_prune	'/v1.49/containers/prune'	post
container_rename	'/v1.49/containers/{id}/rename'	post
container_resize	'/v1.49/containers/{id}/resize'	post
container_restart	'/v1.49/containers/{id}/restart'	post
container_start	'/v1.49/containers/{id}/start'	post
container_stats	'/v1.49/containers/{id}/stats'	get
container_stop	'/v1.49/containers/{id}/stop'	post
container_top	'/v1.49/containers/{id}/top'	get
container_unpause	'/v1.49/containers/{id}/unpause'	post
container_update	'/v1.49/containers/{id}/update'	post
container_wait	'/v1.49/containers/{id}/wait'	post
put_container_archive	'/v1.49/containers/{id}/archive'	put

For distribution operations, the following paths and options are defined:

distribution_inspect	'/v1.49/distribution/{name}/json'	get

For exec operations, the following paths and options are defined:

exec_inspect	'/v1.49/exec/{id}/json'	get
exec_resize	'/v1.49/exec/{id}/resize'	post
exec_start	'/v1.49/exec/{id}/start'	post

For image operations, the following paths and options are defined:

image_build	'/v1.49/build'	post
image_commit	'/v1.49/commit'	post
image_create	'/v1.49/images/create'	post
image_delete	'/v1.49/images/{name}'	delete
image_get	'/v1.49/images/{name}/get'	get
image_get_all	'/v1.49/images/get'	get
image_history	'/v1.49/images/{name}/history'	get
image_inspect	'/v1.49/images/{name}/json'	get
image_list	'/v1.49/images/json'	get
image_load	'/v1.49/images/load'	post
image_prune	'/v1.49/images/prune'	post
image_push	'/v1.49/images/{name}/push'	post
image_search	'/v1.49/images/search'	get
image_tag	'/v1.49/images/{name}/tag'	post

For network operations, the following paths and options are defined:

network_connect	'/v1.49/networks/{id}/connect'	post
network_create	'/v1.49/networks/create'	post
network_delete	'/v1.49/networks/{id}'	delete
network_disconnect	'/v1.49/networks/{id}/disconnect'	post
network_inspect	'/v1.49/networks/{id}'	get
network_list	'/v1.49/networks'	get
network_prune	'/v1.49/networks/prune'	post

For node operations, the following paths and options are defined:

node_delete	'/v1.49/nodes/{id}'	delete
node_inspect	'/v1.49/nodes/{id}'	get
node_list	'/v1.49/nodes'	get
node_update	'/v1.49/nodes/{id}/update'	post

For plugin operations, the following paths and options are defined:

plugin_create	'/v1.49/plugins/create'	post
plugin_delete	'/v1.49/plugins/{name}'	delete
plugin_disable	'/v1.49/plugins/{name}/disable'	post
plugin_enable	'/v1.49/plugins/{name}/enable'	post
plugin_inspect	'/v1.49/plugins/{name}/json'	get
plugin_list	'/v1.49/plugins'	get
plugin_pull	'/v1.49/plugins/pull'	post
plugin_push	'/v1.49/plugins/{name}/push'	post
plugin_set	'/v1.49/plugins/{name}/set'	post
plugin_upgrade	'/v1.49/plugins/{name}/upgrade'	post
get_plugin_privileges	'/v1.49/plugins/privileges'	get

secret_create	'/v1.49/secrets/create'	post
secret_delete	'/v1.49/secrets/{id}'	delete
secret_inspect	'/v1.49/secrets/{id}'	get
secret_list	'/v1.49/secrets'	get
secret_update	'/v1.49/secrets/{id}/update'	post

For service operations, the following paths and options are defined:

service_create	'/v1.49/services/create'	post
service_delete	'/v1.49/services/{id}'	delete
service_inspect	'/v1.49/services/{id}'	get
service_list	'/v1.49/services'	get
service_logs	'/v1.49/services/{id}/logs'	get
service_update	'/v1.49/services/{id}/update'	post

For session operations, the following paths and options are defined:

session	'/v1.49/session'	post

For swarm operations, the following paths and options are defined:

swarm_init	'/v1.49/swarm/init'	post
swarm_inspect	'/v1.49/swarm'	get
swarm_join	'/v1.49/swarm/join'	post
swarm_leave	'/v1.49/swarm/leave'	post
swarm_unlock	'/v1.49/swarm/unlock'	post
swarm_unlockkey	'/v1.49/swarm/unlockkey'	get
swarm_update	'/v1.49/swarm/update'	post

For system operations, the following paths and options are defined:

system_auth	'/v1.49/auth'	post
system_data_usage	'/v1.49/system/df'	get
system_events	'/v1.49/events'	get
system_info	'/v1.49/info'	get
system_ping	'/v1.49/_ping'	get
system_ping_head	'/v1.49/_ping'	head
system_version	'/v1.49/version'	get

For task operations, the following paths and options are defined:

task_inspect	'/v1.49/tasks/{id}'	get
task_list	'/v1.49/tasks'	get
task_logs	'/v1.49/tasks/{id}/logs'	get

For volume operations, the following paths and options are defined:

volume_create	'/v1.49/volumes/create'	post
volume_delete	'/v1.49/volumes/{name}'	delete
volume_inspect	'/v1.49/volumes/{name}'	get
volume_list	'/v1.49/volumes'	get
volume_prune	'/v1.49/volumes/prune'	post
volume_update	'/v1.49/volumes/{name}'	put

Arguments

Operation	The operation to perform, which determines the path and method, as well as any additional options.
Path	The path for the operation, which is derived from the Docker API specification.
Options	List of options for the HTTP request, such as `method` and `accept`.

Chapter 9

library(canny/endian): Big- and little-endian grammars

The endian predicates unify big- and little-endian words, longs and long words with lists of octets by applying shifts and masks to correctly align integer values with their endian-specific octet positions. They utilise integer-relational finite domain CLP(FD) predicates in order to implement forward and reverse translation between octets and integers.

Use of CLP allows the DCG clauses to express the integer relations between octets and their integer interpretations implicitly. The constraints simultaneously define a byte in terms of an octet and vice versa.

byte(*?Byte:integer***) //** *[semidet]*

Parses or generates an octet for *Byte*. Bytes are eight bits wide and unify with octets between 0 and 255 inclusive. Fails for octets falling outside this valid range.

Arguments

Byte	value of octet.

big_endian(*?Width:integer, ?Word:integer***) //** *[semidet]*

Unifies big-endian words with octets.

Example as follows: four octets to one big-endian 32-bit word.

```
?- phrase(big_endian(32, A), [4, 3, 2, 1]),
   format('~16r~n', [A]).
4030201
```

little_endian(*?Width:integer, ?Word:integer***) //** *[semidet]*

Unifies little-endian words with octet stream.

Chapter 10

library(canny/exe)

exe(+*Executable*, +*Arguments*, +*Options***)** *[semidet]*

Implements an experimental approach to wrapping `process_create/3` using `concurrent/3`. It operates concurrent pipe reads, pipe writes and process waits. Predicate parameters match `process_create/3` but with a few minor but key improvements. New *Options* terms offer additional enhanced pipe streaming arguments. See partially-enumerated list below.

- `stdin(codes(Codes))`
- `stdin(atom(Atom))`
- `stdin(string(String))`
- `stdout(codes(Codes))`
- `stdout(atom(Atom))`
- `stdout(string(String))`
- `stderr(codes(Codes))`
- `stderr(atom(Atom))`
- `stderr(string(String))`
- `status(Status)`

If *Options* specifies any of the above terms, `exe/3` prepares goals to write, read and wait concurrently as necessary according to the required configuration. This implies that reading standard output and waiting for the process status happens at the same time. Same goes for writing to standard input. The number of concurrent threads therefore exactly matches the number of concurrent process goals. This goes for clean-up goals as well. Predicate `concurrent/3` does not allow zero threads however; it throws a `type_error`. The implementation always assigns at least one thread which amounts to reusing the calling thread non-concurrently.

All the `std` terms above can also take a stream options list, so can override default encoding on the process pipes. The following example illustrates. It sends a friendly "hello" in Mandarin Chinese through the Unix `tee` command which relays the stream to standard output and tees it off to `/dev/stderr` or standard error for that process. Note that `exe/3` decodes the output and error separately, one as an atom but the other as a string.

```
exe(path(tee),
    [ '/dev/stderr'
    ],
    [ stdin(atom(你好, [encoding(utf8)])),
      stdout(atom(A, [encoding(utf8)])),
      stderr(string(B, [encoding(utf8)])),
      status(exit(0))
    ]).
```

10.0.1 Implementation Notes

Important to close the input stream immediately after writing and during the call phase. Do **not** wait for the clean-up phase to close the input stream, otherwise the process will never terminate. It will hang while waiting for standard input to close, assuming the sub-process reads the input.

This leads to a key caveat when using a single concurrent thread. A single callee thread executes the primary read-write goals in sequential order. The current implementation preserves the *Options* ordering. Hence output should always preceed input, i.e. writing to standard input should go first before attempting to read from standard output. Otherwise the sequence will block indefinitely. For this reason, the number of concurrent threads matches the number of concurrent goals. This abviates the sequencing of the goals because all goals implicitly execute concurrently.

> **To be done** Take care when using the `status(Status)` option unless you have `stdin(null)` on Windows because, for some sub-processes, the goals never complete.

Chapter 11

library(canny/files)

absolute_directory(+*Absolute*, -*Directory***)** *[nondet]*

Finds the directories of *Absolute* by walking up the absolute path until it reaches the root. Operates on paths only; it does not check that *Absolute* actually exists. *Absolute* can be a directory or file path.

Fails if *Absolute* is not an absolute file name, according to `is_absolute_file_name/2`. Works correctly for Unix and Windows paths. However, it finally unifies with the drive letter under Windows, and the root directory (/) on Unix.

Arguments

Absolute	specifies an absolute path name. On Windows it must typically include a driver letter, else not absolute in the complete sense under Microsoft Windows since its file system supports multiple root directories on different mounted drives.

Chapter 12

library(canny/hdx)

hdx(+*StreamPair*, +*Term*, -*Codes*, +*TimeOut***)** *[semidet]*
hdx(+*In*, -*Codes*, +*TimeOut***)** *[semidet]*
hdx(+*Out*, +*Term***)** *[semidet]*

Performs a single half-duplex stream interaction with *StreamPair*. Flushes *Term* to the output stream. Reads pending *Codes* from the input stream within *TimeOut* seconds. Succeeds when a write-read cycle completes without timing out; fails on time-out expiry.

Filling a stream buffer blocks the calling thread if there is no input ready. Pending read operations also block for the same reason. Hence the `wait_for_input/3` **must** precede them.

Arguments

StreamPair	connection from client to server, a closely-associated input and output stream pairing used for half-duplex communication.
Term	to write and flush.
Codes	waited for and extracted from the pending input stream.
TimeOut	in seconds.

Chapter 13

library(canny/maths)

frem(+*X:number*, +*Y:number*, -*Z:number*)** *[det]*

Z is the remainder after dividing *X* by *Y*, calculated by *X* - N * *Y* where N is the nearest integral to *X* / *Y*.

fmod(+*X:number*, +*Y:number*, -*Z:number*)** *[det]*

Z is the remainder after dividing *X* by *Y*, equal to *X* - N * *Y* where N is *X* over *Y* after truncating its fractional part.

epsilon_equal(+*X:number*, +*Y:number*)** *[semidet]*

epsilon_equal(+*Epsilons:number*, +*X:number*, +*Y:number*)** *[semidet]*

Succeeds only when the absolute difference between the two given numbers *X* and *Y* is less than or equal to epsilon, or some factor (*Epsilons*) of epsilon according to rounding limitations.

frexp(+*X:number*, -*Y:number*, -*Exp:integer*)** *[det]*

Answers mantissa *Y* and exponent *Exp* for floating-point number *X*.

Arguments

Y	is the floating-point mantissa falling within the interval [0.5, 1.0). Note the non-inclusive upper bound.

ldexp(+*X:number*, -*Y:number*, +*Exp:integer*)** *[det]*

Loads exponent. Multiplies *X* by 2 to the power *Exp* giving *Y*. Mimics the C math `ldexp(x, exp)` function.

Uses an unusual argument order. Ordering aligns *X*, *Y* and *Exp* with `frexp/3`. Uses `**` rather than `^` operator. *Exp* is an integer.

Arguments

X	is some floating-point value.
Y	is *X* times 2 to the power *Exp*.
Exp	is the exponent, typically an integer.

Chapter 14

library(canny/octet)

octet_bits(*?Octet:integer, ?Fields:list***)** *[semidet]*

Unifies integral eight-bit *Octet* with a list of Value:Width terms where the Width integers sum to eight and the Value terms unify with the shifted bit values encoded within the eight-bit byte.

Arguments

Octet	an eight-bit byte by another name.
Fields	colon-separated value-width terms. The shifted value of the bits comes first before the colon followed by its integer bit width. The list of terms *specify* an octet by sub-spans of bits, or bit *fields*.

Chapter 15

library(canny/pack)

load_pack_modules(+*Pack*, -*Modules*)** *[semidet]*

Finds and loads all Prolog module sources for *Pack*. Also loads test files having once loaded the pack. *Modules* becomes a list of successfully-loaded pack modules.

load_prolog_module(+*Directory*, -*Module*)** *[nondet]*

Loads Prolog source recursively at *Directory* for *Module*. Does **not** load non-module sources, e.g. scripts without a module. Operates non-deterministically for *Module*. Finds and loads all the modules within a given directory; typically amounts to a pack root directory. You can find the File from which the module loaded using module properties, i.e. `module_property(Module, file(File))`.

Chapter 16

library(canny/payloads): Local Payloads

Apply and Property terms must be non-variable. The list below indicates the valid forms of Apply, indicating determinism. Note that only peek and pop perform non-deterministically for all thread-local payloads.

- `reset` is det
- `push` is semi-det
- `peek(Payload)` is non-det
- `pop(Payload)` is non-det
- `[Apply0|Applies]` is semi-det
- *Apply* is semi-det for payload

Properties as follows.

- `top(Property)` is semi-det for payload
- *Property* is semi-det for payload

The first form `top/1` peeks at the latest payload once. It behaves semi-deterministically for the top-most payload.

payload(:*PI*) *[det]*

Makes public multi-file apply-to and property-of predicates using the predicate indicator *PI* of the form M:Payload/{ToArity, OfArity} where arity specifications define the arity or arities for a payload. Defines predicates M:apply_to_Payload/ToArity and M:property_of_Payload/OfArity for module M. Allows comma-separated lists of arities.

apply_to(+*Apply*, :*To*) *[nondet]*
apply_to(+*Applies*, :*To*) *[semidet]*

Arguments	
Applies	is a list of *Apply* terms. It succeeds when all its *Apply* terms succeed, and fails when the first one fails, possibly leaving side effects if the apply-to predicate generates addition effects; though typically not for mutation arity-3 apply-to predicates.

property_of(+*Property, :Of***)** *[nondet]*

Finds *Property* of some payload where the second argument M:*Of* defines the module M and payload atom *Of*.

Property `top/1` peeks semi-deterministically at the top-most payload for some given property.

Chapter 17

library(canny/permutations)

permute_sum_of_int(+*N:nonneg*, -*Integers:list(integer)*)** *[nondet]*

Permute sum. Non-deterministically finds all combinations of integer sums between 1 and *N*. Assumes that 0<=*N*. The number of possible permutations amounts to 2-to-the-power of *N*-1; for *N*=3 there are four as follows: 1+1+1, 1+2, 2+1 and 3.

permute_list_to_grid(+*List0:list*, -*List:list(list)*)** *[nondet]*

Permutes a list to two-dimensional grid, a list of lists. Given an ordered *List0* of elements, unifies *List* with all possible rows of columns. Given `a`, `b` and `c` for example, permutes three rows of single columns `a`, `b`, `c`; then `a` in the first row with `b` and `c` in the second; then `a` and `b` in the first row, `c` alone in the second; finally permutes `a`, `b`, `c` on a single row. Permutations always preserve the order of elements from first to last.

Chapter 18

library(canny/placeholders): Formatting Placeholders

author Roy Ratcliffe
version 0.1.0

This module provides predicates for formatting strings with placeholders. Placeholders are specified in the form of `{name}` within a format string. The placeholders are replaced with corresponding values from a list of options, where each option is specified as `name(Value)`. The result is an atom containing the formatted string. The module uses DCG rules to parse the format string and replace the placeholders with the corresponding values.

The main predicate is `format_placeholders/3`, which takes a format string, an atom to hold the result, and a list of options. It processes the format string, replacing placeholders with their corresponding values from the options list. If a placeholder does not have a corresponding value, it will fail.

The `format_placeholders/3` predicate formats a string with placeholders, while `format_placeholders/4` allows for additional options to be returned; namely, the remaining options after processing the placeholders.

format_placeholders(+*Format, -Atom, +Options***)** *[det]*
format_placeholders(+*Format, -Atom, +Options, -RestOptions***)** *[det]*

Formats a string with placeholders in the form of `{name}`. The placeholders are replaced with corresponding values from the options list. The result is an atom with the formatted string.

The *Format* string can be any atom or string containing placeholders. The *Options* list should contain terms of the form `name(Value)`, where `name` is the placeholder name and *Value* is the value to replace it with. If a placeholder does not have a corresponding value in the *Options* list, it will not be replaced, and the placeholder will remain in the resulting atom.

Arguments

Format	The format string containing placeholders.
Atom	The resulting atom with placeholders replaced.
Options	The list of options containing values for placeholders.
RestOptions	The remaining options after processing the placeholders.

placeholders(-*Terms*, *?Options***) //** *[det]*

Formats a list of terms by replacing placeholders in the form of `{name}` with corresponding values from the options list. The placeholders are replaced with the values associated with the names in the options list.

The result is a list of atoms and values, and a completed options list.

Arguments

Terms	The list of terms to be formatted.
Options	The list of options containing values for placeholders.

placeholders(+*Terms0*, -*Terms*, +*Options0*, -*Options***) //** *[det]*

Processes a format string with placeholders using a list of terms and options. The format string is the difference list of codes, where placeholders are replaced with values from the options list. The result is a list of atomics and an updated options list.

Scans the input, replacing placeholders of the form `{name}` with values from the options list. The result is a list of atoms and values, and an updated options list. Uses DCG rules for flexible parsing and substitution.

The resulting list of terms contains atoms and values, where each placeholder is replaced with the corresponding value from the options list. The options list is updated to include any new options found in the format string.

Unifies the same placeholder with the same value in the options list if it appears more than once. Placeholders can appear in the format string multiple times, and each occurrence will be replaced with the same value.

Arguments

Terms0	The initial list of terms to be processed.
Terms	The resulting list of terms after processing.
Options0	The initial list of options to be processed.
Options	The resulting list of options after processing.

Chapter 19

library(canny/pop)

pop_lsbs(+*A:nonneg*, -*L:list***)** *[det]*

Unifies non-negative integer *A* with its set bits *L* in least-significate priority order. Defined only for non-negative *A*. Throws a domain error otherwise.

Errors `domain_error(not_less_than_one, A)` if *A* less than 0.

Chapter 20

library(canny/redis)

redis_last_streams(+*Reads*, -*Streams:list***)** *[det]*
redis_last_streams(+*Reads*, ?*Tag*, -*Streams:dict***)** *[det]*
Collates the last *Streams* for a given list of *Reads*, the reply from an XREAD command. The implementation assumes that each stream's read reply has one entry at least, else the stream does not present a reply.

redis_last_stream_entry(+*Entries*, -*StreamId*, -*Fields***)** *[semidet]*
redis_last_stream_entry(+*Entries:list(list)*, -*StreamId:atom*, ?*Tag:atom*, -*Fields:dict***)** *[semidet]*
Unifies with the last *StreamId* and *Fields*. It fails for empty *Entries*. Each entry comprises a *StreamId* and a set of *Fields*.

redis_keys_and_stream_ids(+*Streams*, ?*Tag*, -*Keys*, -*StreamIds***)** *[det]*
redis_keys_and_stream_ids(+*Pairs*, -*Keys*, -*StreamIds***)** *[det]*
Streams or *Pairs* of *Keys* and *StreamIds*. Arity-3 exists with *Tag* in order to unify with a dictionary by *Tag*.

Arguments

Streams	is a dictionary of stream identifiers, indexed by stream key.
Keys	is a list of stream keys.
StreamIds	is a list of corrected stream identifiers. The predicate applies `redis_stream_id/3` to the incoming identifiers, allowing for arbitrary milliseconds-sequence pairs including implied missing zero sequence number.

redis_stream_read(+*Reads*, -*Key*, -*StreamId*, -*Fields***)** *[nondet]*
redis_stream_read(+*Reads*, -*Key*, -*StreamId*, ?*Tag*, -*Fields***)** *[nondet]*
Unifies with all *Key*, *StreamId* and array of *Fields* for all *Reads*.

Arguments

Reads	is a list of [*Key*, Entries] lists, a list of lists. The sub-lists always have two items: the *Key* of the stream followed by another sub-list of stream entries.

redis_stream_entry(+*Entries*, -*StreamId*, -*Fields***)** *[nondet]*
redis_stream_entry(+*Entries:list*, -*StreamId:pair(nonneg,nonneg)*, ?*Tag:atom*, -*Fields:dict***)** *[nondet]*
redis_stream_entry(+*Reads:list*, -*Key:atom*, -*StreamId:pair(nonneg,nonneg)*, ?*Tag:atom*, -*Fields:dict***)** *[n*

Unifies non-deterministically with all *Entries*, or *Fields* dictionaries embedded with multi-stream *Reads*. Decodes the stream identifier and the Entry.

Arguments	
Entries	is a list of [*StreamId*, *Fields*] lists, another list of lists. Each sub-list describes an "entry" within the stream, a pairing between an identifier and some fields.

redis_stream_id(*?RedisTimeSeqPair***)** *[semidet]*
redis_stream_id(*?StreamId:text, ?RedisTimeSeqPair***)** *[semidet]*
redis_stream_id(*?StreamId:text, ?RedisTime:nonneg, ?Seq:nonneg***)** *[semidet]*

Stream identifier to millisecond and sequence numbers. In practice, the numbers always convert to integers.

Deliberately validates incoming Redis time and sequence numbers. Both must be integers and both must be zero or more. The predicates fail otherwise. Internally, Redis stores stream identifiers as 128-bit unsigned integers split in half for the time and sequence values, each of 64 bits.

The 3-arity version of the predicate handles extraction of time and sequence integers from arbitrary stream identifiers: text or compound terms, including implied zero-sequence stream identifier with a single non-negative integer representing a millisecond Unix time.

Arguments	
StreamId	identifies a stream message or entry, element or item. All these terms apply to the contents of a stream, but Redis internally refers to the content as *entries*.
RedisTimeSeqPair	is a pair of non-negative integers, time and sequence. The Redis time equals Unix time multiplied by 1,000; in other words, Unix time in milliseconds.

redis_time(+*RedisTime***)** *[semidet]*

Successful when *RedisTime* is a positive integer. Redis times amount to millisecond-scale Unix times.

Arguments	
RedisTime	in milliseconds since 1970.

redis_date_time(+*RedisTime, -DateTime, +TimeZone***)** *[det]*

Converts *RedisTime* to *DateTime* within *TimeZone*.

Chapter 21

library(canny/redis_streams)

xrange(+*Redis*, +*Key:atom*, -*Entries:list*, +*Options:list***)** *[det]*

Applies range selection to *Key* stream. *Options* optionally specify the start and end stream identifiers, defaulting to - and + respectively or in reverse if `rev(true)` included in *Options* list; the plus stream identifier stands for the maximum identifier, or the newest, whereas the minus identifier stands for the oldest. Option `count(Count)` limits the number of entries to read by *Count* items.

The following always unifies *Entries* with `[]`.

```
xrange(Server, Key, Entries, [start(+)]).
xrange(Server, Key, Entries, [rev(true), start(-)]).
```

xread(+*Redis*, +*Streams:dict*, -*Reads:list*, +*Options:list***)** *[semidet]*

Unifies *Reads* from *Streams*. Fails on time-out, if option `block(Milliseconds)` specifies a non-zero blocking delay.

Arguments

Reads	by stream key. The reply has the form [Key, Entries] for each stream where each member of Entries has the form [StreamID, Fields] where Fields is an array of keys and values.

xread_call(+*Redis*, +*Streams*, :*Goal*, -*Fields*, +*Options***)** *[semidet]*
xread_call(+*Redis*, +*Streams*, :*Goal*, ?*Tag*, -*Fields*, +*Options***)** *[semidet]*

Reads *Streams* continuously until *Goal* succeeds or times out. Also supports a *Redis* time limit option so that blocking, if used, does not continue indefinately even on a very busy stream set. The limit applies to any of the given streams; it acts as a time threshold for continuous blocking failures.

Chapter 22

library(canny/shifter)

bit_shift(+*Shifter*, *?Left*, *?Right***)** *[semidet]*

Shifts bits left or right depending on the argument mode. Mode (+, -, +) shifts left whereas mode (+, +, -) shifts right. The first argument specifies the position of the bit or bits in *Left*, the second argument, while the third argument specifies the aligned *Right* bits. The shift moves in the direction of the variable argument, towards the (-) mode argument.

The *Shifter* argument provides three different ways to specify a bit shift and bit width: either by an exclusive range using + and – terms; or an *inclusive* range using : terms; or finally just a single bit shift which implies a width of one bit. Colons operate inclusively whereas plus and minus apply exclusive upper ranges.

It first finds the amount of Shift required and the bit Width. After computing the lefthand and righthand bit masks, it finally performs a shift-mask or mask-shift for left and right shifts respectively.

Arguments

Shifter	is a Shift+Width, Shift-Width, High:Low, Low:High or just a single integer Shift for single bits.
Left	is the left-shifted integer.
Right	is the right-shifted integer.

Chapter 23

library(canny/situations)

situation_apply(*?Situation:any, ?Apply***)** *[nondet]*

Mutates *Situation*. *Apply* term to *Situation*, where *Apply* is one of the following. Note that the *Apply* term may be nonground. It can contain variables if the situation mutation generates new information.

module(*?Module***)**

Sets up *Situation* using *Module*. Establishes the dynamic predicate options for the temporary situation module used for persisting situation Now-At and Was-When tuples.

An important side effect occurs for ground *Situation* terms. The implementation creates the situation's temporary module and applies default options to its new dynamic predicates. The `module(Module)` term unifies with the newly-created or existing situation module.

The predicate's determinism collapses to semi-determinism for ground situations. Otherwise with variable *Situation* components, the predicate unifies with all matching situations, unifying with `module(Module)` nondeterministically.

now(*+Now:any***)**

now(*+Now:any, +At:number***)**

Makes some *Situation* become *Now* for time index *At*, at the next fixation. Effectively schedules a pending update one or more times; the next situation `fix/0` fixes the pending situation changes at some future point. The `now/1` form applies *Now* to *Situation* at the current Unix epoch time.

Uses `canny:apply_to_situation/2` when *Situation* is ground, but uses `canny:property_of_situation/2` otherwise. Asserts therefore for multiple situations if *Situation* comprises variables. You cannot therefore have nonground situations.

fix

fix(*+Now:any***)**

Fixating situations does three important things. First, it adds new Previous-When pairs to the situation history. They become `was/2` dynamic facts (clauses without rules). Second, it adds, replaces or removes the

most current Current-When pair. This allows detection of non-events, e.g. when something disappears. Some types of situation might require such event edges. Finally, fixating broadcasts situation-change messages.

The rule for fixing the Current-When pair goes like this: Is there a new `now/2`, at least one? The latest becomes the new current. Any others become Previous-When. If there is no `now/2`, then the current disappears. Messages broadcast accordingly. If there is more than one `now/2`, only the latest becomes current. Hence currently-previously only transitions once in-between fixations.

Term `fix/1` is a shortcut for `now(Now, At)` and `fix` where `At` becomes the current Unix epoch time. Fixes but does not retract history terms.

retract(+*When:number*)

retract(?*When:number*, +*Delay:number*)

Retracts all `was/2` clauses for all matching *Situation* terms. Term `retract(_, Delay)` retracts all `was/2` history terms using the last term's latest time stamp. In this way, you can retract situations without knowing their absolute time. For example, you can retract everything older than 60 seconds from the last known history term when you `retract(_, 60)`.

The second argument *Apply* can be a list of terms to apply, including nested lists of terms. All terms apply in order first to last, and depth first.

Arguments

Now	is the state of a *Situation* at some point in time. The Now term must be non-variable but not necessarily ground. Dictionaries with unbound tags can exist within the situation calculus.

situation_property(?*Situation:any*, ?*Property*) *[nondet]*

Property of *Situation*.

module(?*Module*)

Marries situation terms with universally-unique modules, one for one. All dynamic situations link a situation term with a module. This design addresses performance. Retracts take a long time, relatively, especially for dynamic predicates with very many clauses; upwards of 10,000 clauses for example. Note, you can never delete the situation-module association, but you can retract all the dynamic clauses belonging to a situation.

defined

Situation is defined whenever a unique situation module already exists for the given *Situation*. Amounts to the same as asking for `module(_)` property.

currently(?*Current:any*)

currently(?*Current:any*, ?*When:number*)

currently(*Current:any*, *for(Seconds:number)*)

Unifies with *Current* for *Situation* and When it happened. Unifies with the

one and only *Current* state for all the matching *Situation* terms. Unifies non-deterministically for all *Situation* solutions, but semi-deterministically for *Current* state. Thus allows for multiple matching situations but only one *Current* solution.

You can replace the When term with `for(Seconds)` in order to measure elapsed interval since fixing *Situation*. Same applies to `previously/2` except that the current situation time stamp serves as the baseline time, else defaults to the current time.

previously(*?Previous:any***)**

previously(*?Previous:any, ?When:number***)**

previously(*Previous:any, for(Seconds:number)***)**

Finds *Previous* state of *Situation*, non-deterministically resolving zero or more matching *Situation* terms. Fails if no previous *Situation* condition.

history(*?History:list(compound)***)**

Unifies *History* with all current and previous situation conditions, including their time stamps. *History* is a sequence of compounds of the form `was(Was, When)` where *Situation* is effectively a primitive condition coordinate, Was is a sensing outcome and When marks the moment that the outcome transpired.

Chapter 24

library(canny/situations_debugging)

print_situation_history_lengths *[det]*

Finds all situations. Samples their histories and measures the history lengths. Uses = when sorting; do not remove duplicates. Prints a table of situations by their history length, longest history comes first. Filters out single-element histories for the sake of noise minimisation.

Chapter 25

library(canny/z)

enz(+*Data:list*, +*File*) *[semidet]*

Zips *Data* to *File*. Writes `zip(Name:atom, Info:dict, MemFile:memory_file)` functor triples to *File* where *Name* is the key; *MemFile* is the content as a memory file. Converts the *Info* dictionary to new-member options when building up the zipper. Ignores any non-valid key pairs, including offset plus compressed and uncompressed sizes.

The implementation *asserts* octet encoding for new files with a zipper. The predicate for creating a zipper member does **not** allow for an encoding option. It encodes as binary by default.

unz(+*File*, -*Data:list*) *[semidet]*

Unzips *File* to *Data*, a list of `zip` functors with *Name* atom, *Info* dictionary and *MemFile* content arguments.

You cannot apply `unz/2` to an empty zip *File*. A bug crashes the entire Prolog run-time virtual machine.

Chapter 26

library(data/frame)

columns_to_rows(*?ListOfColumns, ?ListOfRows***)** *[semidet]*

Transforms *ListOfColumns* to *ListOfRows*, where a row is a list of key-value pairs, one for each cell. By example,

```
[a=[1, 2], b=[3, 4]]
```

becomes

```
[[a-1, b-3], [a-2, b-4]]
```

Else fails if rows or columns do not match. The output list of lists suitably conforms to `dict_create/3` Data payloads from which you can build dictionaries.

```
?- columns_to_rows([a=[1, 2], b=[3, 4]], A),
   maplist([B, C]>>dict_create(C, row, B), A, D).
A = [[a-1, b-3], [a-2, b-4]],
D = [row{a:1, b:3}, row{a:2, b:4}].
```

Chapter 27

library(dcg/endian)

endian(*?BigOrLittle, ?Width, ?Value***) //** *[semidet]*

Applies `big` or `little`-endian ordering grammar to an integer *Value* of any *Width*.

Divides the problem in two: firstly the 'endianness' span which unifies an input or output phrase with the bit width of a value, and secondly the shifted bitwise-OR phase that translates between coded eight-bit octets and un-encoded integers of unlimited bit width by accumulation.

Arguments	
BigOrLittle	is the atom `big` or `little` specifying the endianness of the coded *Value*.
Width	is the multiple-of-eight bit width of the endian-ordered octet phrase.
Value	is the un-encoded integer value of unlimited bit width.

big_endian(*?Width, ?Value***) //** *[semidet]*

Implements the grammar for `endian(big, Width, Value)` super-grammar.

In (-, +) mode the accumulator recurses *first* and then the residual *Value_* merges with the accumulated *Value* because the first octet code is the most-significant byte of the value for big-endian integer representations, rather than the least-significant. The `0 =< H, H =< 255` guard conditions ensure failure for non-octet code items in the list.

little_endian(*?Width, ?Value***) //** *[semidet]*

Implements `endian(little, Width, Value)` grammar.

Little-endian accumulators perform the same logical unification as for big-endian only in reverse. The only difference between big and little: recurse first or recurse last. Apart from that subtle but essential difference, the inner computation behaves identically.

Chapter 28

library(dcg/files)

directory_entry(+*Directory, ?Entry*) // *[nondet]*

Neatly traverses a file system using a grammar.

Finds files and skips the special dot entries. Here, *Entry* refers to a file. The grammar recursively traverses sub-directories beneath the given *Directory* and yields every existing file path at *Entry*. The directory acts as the root of the scan; it joins with the entry to yield the full path of the file, but **not** with the difference list. The second *List* argument of `phrase/2` unifies with a list of the corresponding sub-path components **without** the root. The caller sees the full path **and** the relative sub-components.

Note that the second clause appears in the DCG expanded form with the two hidden arguments: the pre-parsed input list *S0* and the post-parsed output list *S*. For non-directory entries, the input list unifies with nil `[]` because it represents a terminal node in the directory tree, and the post-parsed terms amount to the accumulated *Entries* spanning the sub-directory entries in-between the original root directory and the file itself.

directory_entry(+*Directory, ?Entry*) *[nondet]*

Finds files and directories in the *Directory* except special files: dot, the current directory; and double dot, the parent directory.

No need to check if the *Entry* exists. It does exist at the time of directory iteration. That could easily change by deleting, moving or renaming the entry.

Chapter 29

library(doc/latex)

latex_for_pack(+*Spec*, +*OutFile*, +*Options***)** *[det]*

Chapter 30

library(docker/random_names)

random_name(*?Name***)** *[nondet]*

Non-deterministically generates Docker-style random names. Uses `random_permutation/2` and `member/2`, rather than `random_member/2`, in order to generate all possible random names by back-tracking if necessary.

The engine-based implementation has two key features: generates random permutations of both left and right sub-names independently; does not repeat until after unifying all permutations. This implies that two consecutive names will never be the same up until the boundary event between two consecutive randomisations. There is a possibility, albeit small, that the last random name from one sequence might accidentally match the first name in the next random sequence. There are 23,500 possible combinations.

The implementation is **not** the most efficient, but does perform accurate randomisation over all left-right name permutations.

Allows *Name* to collapse to semi-determinism with ground terms without continuous random-name generation since it will never match an atom that does not belong to the Docker-random name set. The engine-based non-determinism only kicks in when *Name* unbound.

random_name_chk(*-Name:atom***)** *[det]*

Generates a random *Name*.

Only ever fails if *Name* is bound and fails to match the next random *Name*, without testing for an unbound argument. That makes little sense, so fails unless *Name* is a variable.

random_name_chk(*?LHS:atom, ?RHS:atom***)** *[semidet]*

Unifies *LHS-RHS* with one random name, a randomised selection from all possible names.

Note, this does **not** naturally work in (+, ?) or (?, +) or (+, +) modes, even if required. Predicate `random_member/2` fails semi-deterministically if the given atom fails to match the randomised selection. Unifies semi-deterministically for ground atoms in order to work correctly for non-variable arguments. It collapses to failure if the argument cannot unify with random-name possibilities.

Chapter 31

library(gh/api): GitHub API

author Roy Ratcliffe

You need a personal access token for updates. You do **not** require them for public access.

ghapi_update_gist(+*GistID*, +*Data*, -*Reply*, +*Options***)** *[det]*

Updates a Gist by its unique identifier. *Data* is the patch payload as a JSON object, or dictionary if you include `json_object(dict)` in *Options*. *Reply* is the updated Gist in JSON on success.

The example below illustrates a Gist update using a JSON term. Notice the doubly-nested `json/1` terms. The first sets up the HTTP request for JSON while the inner term specifies a JSON *object* payload. In this example, the update adds or replaces the `cov.json` file with content of "{}" as serialised JSON. Update requests for Gists have a `files` object with a nested filename-object comprising a content string for the new contents of the file.

```
ghapi_update_gist(
    ec92ac84832950815861d35c2f661953,
    json(json([ files=json([ 'cov.json'=json([ content='{}'
                                            ])
                           ])
            ])), _, []).
```

See also `https://docs.github.com/en/rest/reference/gists#update-a-gist`

ghapi_get(+*PathComponents*, +*Data*, +*Options***)** *[det]*

Accesses the GitHub API. Supports JSON terms and dictionaries. For example, the following goal accesses the GitHub Gist API looking for a particular Gist by its identifier and unifies *A* with a JSON term representing the Gist's current contents and state.

```
ghapi_get([gists, ec92ac84832950815861d35c2f661953], A, []).
```

Supports all HTTP methods despite the predicate name. The "get" mirrors the underlying `http_get/3` method which also supports all methods. POST and PATCH send data using the `post/1` option and override the default HTTP verb using the `method/1` option. Similarly here.

Handles authentication via settings, and from the system environment indirectly. Option `ghapi_access_token/1` overrides both. Order of overriding proceeds as: option, setting, environment, none. Empty atom counts as none.

Abstracts away the path using path components. Argument *PathComponents* is an atomic list specifying the URL path.

Chapter 32

library(html/scrapes)

scrape_row(+*URL*, -*Row*)** *[nondet]*

Scrapes all table rows non-deterministically by row within each table. Tables must have table headers, `thead` elements.

Scrapes distinct rows. Distinct is important because HTML documents contain tables within tables within tables. Attempts to permit some flexibility. Asking for sub-rows finds head sub-rows; catches and filters out by disunifying data with heads.

Chapter 33

library(ieee/754)

ieee_754_float(+*Bits, ?Word, ?Float*) *[det]*
ieee_754_float(-*Bits, ?Word, ?Float*) *[nondet]*

Performs two-way pack and unpack for IEEE 754 floating-point numbers represented as words.

Not designed for performance. Uses CLP(FD) for bit manipulation. and hence remains within the integer domain. *Float* arithmetic applies outside the finite-domain constraints.

Arguments

Word	is a non-negative integer. This implementation does not handle negative integers. Negative support implies a non-determinate solution for packing. A positive and negative answer exists for any given *Float*.
Sig	is the floating-point significand between plus and minus 1. Uses Sig rather than Mantissa; Sig short for Significand, another word for mantissa.

Chapter 34

library(linear/algebra): Linear algebra

"The introduction of numbers as coordinates is an act of violence."–Hermann Weyl, 1885-1955.

Vectors are just lists of numbers, or scalars. These scalars apply to arbitrary abstract dimensions. For example, a two-dimensional vector [1, 2] applies two scalars, 1 and 2, to dimensional units *i* and *j*; known as the basis vectors for the coordinate system.

Is it possible, advisable, sensible to describe vector and matrix operations using Constraint Logic Programming (CLP) techniques? That is, since vectors and matrices are basically columns and rows of real-numeric scalars, their operators amount to constrained relationships between real numbers and hence open to the application of CLP over reals. The simple answer is yes, the linear_algebra predicates let you express vector operators using real-number constraints.

Constraint logic adds some important features to vector operations. Suppose for instance that you have a simple addition of two vectors, a vector translation of U+V=W. Add U to V giving W. The following statements all hold true. Note that the CLP-based translation unifies correctly when W is unknown but also when U or V is unknown. Given any two, you can ask for the missing vector.

```
?- vector_translate([1, 1], [2, 2], W).
W = [3.0, 3.0] ;
false.
?- vector_translate([1, 1], V, [3, 3]).
V = [2.0, 2.0] ;
false.
?- vector_translate(U, [2, 2], [3, 3]).
U = [1.0, 1.0] ;
false.
```

Note also that the predicate answers non-deterministically with back-tracking until no alternative answer exists. This presumes that alternatives could exist at least in theory if not in practice. Trailing choice-points remain unless you cut them.

matrix_dimensions(*?Matrix:list(list(number)), ?Rows:nonneg, ?Columns:nonneg***)** *[semidet]*
Dimensions of *Matrix* where dimensions are *Rows* and *Columns*.

A matrix of M rows and N columns is an M-by-N matrix. A matrix with a single row is a row vector; one with a single column is a column vector. Because the linear_algebra module uses lists to represent vectors and matrices, you need never distinguish between row and column vectors.

Boundary cases exist. The dimensions of an empty matrix [] equals [0, _] rather than [0, 0]. And this works in reverse; the matrix unifying with dimensions [0, _] equals [].

matrix_identity(*+Order:nonneg, -Matrix:list(list(number))***)** *[semidet]*
Matrix becomes an identity matrix of *Order* dimensions. The result is a square diagonal matrix of *Order* rows and *Order* columns.

The first list of scalars (call it a row or column) becomes 1 followed by *Order*-1 zeros. Subsequent scalar elements become an *Order*-1 identity matrix with a 0-scalar prefix for every sub-list. Operates recursively albeit without tail recursion.

Fails when matrix size *Order* is less than zero.

matrix_transpose(*?Matrix0:list(list(number)), ?Matrix:list(list(number))***)** *[semidet]*
Transposes matrices. The matrix is a list of lists. Fails unless all the sub-lists share the same length. Works in both directions, and works with non-numerical elements. Only operates at the level of two-dimensional lists, a list with sub-lists. Sub-sub-lists remain lists and un-transposed if sub-lists comprise list elements.

matrix_rotation(*?Theta:number, ?Matrix:list(list(number))***)** *[nondet]*
The constructed matrix applies to column vectors [X, Y] where positive *Theta* rotates X and Y anticlockwise; negative rotates clockwise. Transpose the rotation matrix to reverse the angle of rotation; positive for clockwise, negative anticlockwise.

vector_distance(*?V:list(number), ?Distance:number***)** *[semidet]*
vector_distance(*?U:list(number), ?V:list(number), ?Distance:number***)** *[semidet]*
Distance of the vector *V* from its origin. *Distance* is Euclidean distance between two vectors where the first vector is the origin. Note that Euclidean is just one of many distances, including Manhattan and chessboard, etc. The predicate is called distance, rather than length. The term length overloads on the dimension of a vector, its number of numeric elements.

vector_translate(*?U, ?V, ?W***)** *[nondet]*
Translation works forwards and backwards. Since *U*+*V*=*W* it follows that *U*=*W*-*V* and also *V*=*W*-*U*. So for unbound *U*, the vector becomes *W*-*V* and similarly for *V*.

vector_scale(*?Scalar:number, ?U:list(number), ?V:list(number)***)** *[nondet]*
Vector *U* scales by *Scalar* to *V*.

What is the difference between multiply and scale? Multiplication multiplies two vectors whereas scaling multiplies a vector by a scalar; hence the verb to scale. Why is the scalar at the front of the argument list? This allows the meta-call of `vector_scale(Scalar)` passing two vector arguments, e.g. when mapping lists of vectors.

The implementation performs non-deterministically because the CLP(R) library leaves a choice point when searching for alternative arithmetical solutions.

vector_heading(*?V:list(number), ?Heading:number***)** *[semidet]*

Heading in radians of vector *V*. Succeeds only for two-dimensional vectors. Normalises the *Heading* angle in (+, -) mode; negative angles wrap to the range between pi and two-pi. Similarly, normalises the vector *V* in (-, +) mode; *V* has unit length.

scalar_power(*?X:number, ?Y:number, ?Z:number***)** *[nondet]*

Z is *Y* to the power *X*.

The first argument *X* is the exponent rather than *Y*, first rather than second argument. This allows you to curry the predicate by fixing the first exponent argument. In other words, `scalar_power(2, A, B)` squares A to B.

Chapter 35

library(ollama/chat): Ollama Chat

Idiomatic SWI-Prolog HTTP client module for interacting with an Ollama chat API.

35.1 Usage

How to use and abuse the interface? Take some examples. The following queries run with HTTP debugging enabled. Notice the headers.

For streaming:

```
?- ollama_chat([_{role:user, content:"Hello"}], Message, [stream(true)]).
% http_open: Connecting to localhost:11434 ...
%       ok <stream>(000001a4454d2630) ---> <stream>(000001a4454d2740)
% HTTP/1.1 200 OK
% Content-Type: application/x-ndjson
% Date: Sat, 31 May 2025 10:48:49 GMT
% Connection: close
% Transfer-Encoding: chunked
Message = _{content:" Hello", role:"assistant"} ;
Message = _{content:"!", role:"assistant"} ;
Message = _{content:" How", role:"assistant"} ;
Message = _{content:" can", role:"assistant"} ;
Message = _{content:" I", role:"assistant"} ;
Message = _{content:" assist", role:"assistant"} ;
Message = _{content:" you", role:"assistant"} ;
Message = _{content:" today", role:"assistant"} ;
Message = _{content:"?", role:"assistant"} ;
Message = _{content:"", role:"assistant"}.
```

The streaming content type is **not** "application/json" but rather newline-delimited JSON. This is correct. Our addition to the JSON type multifile predicate catches this.

For non-streaming:

```
?- ollama_chat([_{role:user, content:"Hello"}], Message, [stream(false)]).
% http_open: Connecting to localhost:11434 ...
%       ok <stream>(000001a4454d3ea0) ---> <stream>(000001a4454d4e90)
% HTTP/1.1 200 OK
% Content-Type: application/json; charset=utf-8
% Date: Sat, 31 May 2025 10:50:04 GMT
% Content-Length: 347
% Connection: close
Message = _{content:" Sure, I'm here to help! How can I assist you today?", role:"assistant"}.
```

ollama_chat(+*Messages:list(dict), -Message:dict, +Options:list***)** *[nondet]*

Leverages SWI-Prolog's HTTP libraries for Ollama chat API interaction. To stream or not to stream? That becomes an option, specify either `stream(true)` or `stream(false)`, defaulting to streaming. This option selects the predicate's determinism. Predicate `ollama_chat/3` becomes non-deterministic **when** streaming, but falls back to deterministic when not.

Pulls out the message from the reply; it becomes the result of the chat interaction: many messages in, one message out. Taking only the message assumes that the other keys within the reply dictionary have less value. Callers can usually ignore them. The predicate unifies with `reply(Reply)` in the *Options* argument if the caller wants to view the detailed response information.

Assumes that the reply is always a dictionary type without first checking. The implementation relies on the lower-level HTTP layers for parsing and rendering the correct term type. It also assumes that the dictionary always contains a "message" pair. Throws an exception when this presumption fails; this is a design feature because all responses must have a message.

Chapter 36

library(os/apps): Operation system apps

What is an app? In this operating-system `os_apps` module context, simply something you can start and stop using a process. It has no standard input, and typically none or minimal standard output and error.

There is an important distinction between apps and processes. These predicates use processes to launch apps. An application typically has one process instance; else if not, has differing arguments to distinguish one running instance of the app from another. Hence for the same reason, the app model here ignores "standard input." Apps have no such input stream, conceptually speaking.

Is "app" the right word to describe such a thing? English limits the alternatives: process, no because that means something that loads an app; program, no because that generally refers the app's image including its resources.

36.1 App configuration

Apps start by creating a process. Processes have four distinct specification parameter groups: a path specification, a list of arguments, possibly some execution options along with some optional encoding and other run-time related options. Call this the application's configuration.

The `os_apps` predicates rely on multi-file `os:property_for_app/2` to configure the app launch path, arguments and options. The property-for-app predicate supplies an app's configuration non-deterministically using three sub-terms for the first Property argument, as follows.

- `os:property_for_app(path(Path), App)`
- `os:property_for_app(argument(Argument), App)`
- `os:property_for_app(option(Option), App)`

Two things to note about these predicates; (1) App is a compound describing the app **and** its app-specific configuration information; (2) the first Property argument collates arguments and options non-deterministically. Predicate `app_start/1` finds all the argument- and option-solutions *in the order defined*.

36.2 Start up and shut down

By default, starting an app does **not** persist the app. It does not restart if the user or some other agent, including bugs, causes the app to exit. Consequently, this module offers a secondary app-servicing layer. You can start up or shut down any app. This amounts to starting and upping or stopping and downing, but substitutes shut for stop. Starting up issues a start but also watches for stopping.

36.3 Broadcasts

Sends three broadcast messages for any given App, as follows:

- os:app_started(App)
- os:app_decoded(App, stdout(Codes))
- os:app_decoded(App, stderr(Codes))
- os:app_stopped(App, Status)

Running apps send zero or more os:app_decoded(App, Term) messages, one for every line appearing in their standard output and standard error streams. Removes line terminators. App termination broadcasts an exit(Code) term for its final Status.

36.4 Usage

You can start or stop an app.

```
app_start(App)
app_stop(App)
```

App is some compound that identifies which app to start and stop. You define an App using os:property_for_app/2 multi-file predicate. You must at least define an app's path using, as an example:

```
os:property_for_app(path(path(mspaint)), mspaint) :- !.
```

Note that the Path is a path Spec used by process_create/3, so can include a path-relative term as above. This is enough to launch the Microsoft Paint app on Windows. No need for arguments and options for this example. Starting a *running* app does not start a new instance. Rather, it succeeds for the existing instance. The green cut prevents unnecessary backtracking.

You can start and continuously restart apps using app_up/1, and subsequently shut them down with app_down/1.

36.4.1 Apps testing

On a Windows system, try the following for example. It launches Microsoft Paint. Exit the Paint app after `app_up/1` below and it will relaunch automatically.

```
?- [library(os/apps), library(os/apps_testing)].
true.

?- app_up(mspaint).
true.

?- app_down(mspaint).
true.
```

app_property(*?App:compound, ?Property***)** *[nondet]*
Property of *App*.

Note that `app_property(App, defined)` should **not** throw an exception. Some apps have an indeterminate number of invocations where *App* is a compound with variables. Make sure that the necessary properties are ground, rather than unbound.

Collapses non-determinism to determinism by collecting *App* and *Property* pairs before expanding the bag to members non-deterministically.

app_start(*?App:compound***)** *[nondet]*
Starts an *App* if not already running. Starts more than one apps non-deterministically if *App* binds with more than one specifier. Does not restart the app if launching fails. See `app_up/1` for automatic restarts. An app's argument and option properties execute non-deterministically.

Options can include the following:

encoding(*Encoding***)**
an encoding option for the output and error streams.

alias(*Alias***)**
an alias prefix for the detached watcher thread.

Checks for not-running **after** unifying with the *App* path. Succeeds if already running.

app_stop(*?App:compound***)** *[nondet]*
Kills the *App* process. Stopping the app does not prevent subsequent automatic restart.

Killing does **not** retract the `app_pid/2` by design. Doing so would trigger a failure warning. (The waiting PID-monitor thread would die on failure because its retract attempt fails.)

app_up(*?App:compound***)** *[nondet]*

Starts up an *App*.

Semantics of this predicate rely on `app_start/1` succeeding even if already started. That way, you can start an app then subsequently *up* it, meaning stay up. Hence, you can `app_stop(App)` to force a restart if already `app_up(App)`. Stopping an app does not *down* it!

Note that `app_start/1` will fail for one of two reasons: (1) because the *App* has not been defined yet; (2) because starting it fails for some reason.

app_down(*?App:compound***)** *[nondet]*

Shuts down an *App*. Shuts down multiple apps non-deterministically if the *App* compound matches more than one app definition.

Chapter 37

library(os/lc)

lc_r(+*Extensions:list*)**** *[det]*

Recursively counts and prints a table of the number of lines within read-access files having one of the given *Extensions* found in the current directory or one of its sub-directories. Prints the results in line-count descending order with the total count appearing first against an asterisk, standing for all lines counted.

lc_r(-*Pairs*, +*Options*)**** *[det]*

Counts lines in files recursively within the current directory.

lc_r(+*Directory*, -*Pairs*, +*Options*)**** *[det]*

Counts lines within files starting at *Directory*.

lc(+*Directory*, -*Pairs*, +*Options*)**** *[det]*

Counts lines in files starting at *Directory* and using *Options*. Counts for each file concurrently in order to maintain high performance.

Arguments

Pairs	is a list of atom-integer pairs where the relative path of a matching text file is the first pair-element, and the number of lines counted is the second pair-element.

Chapter 38

library(os/search_paths)

search_path_prepend(+*Name:atom*, +*Directory:atom*)** *[det]*

Adds *Directory* to a search-path environment variable. Note, this is not naturally an atomic operation but the prepend makes it thread safe by wrapping the fetching and storing within a mutex.

Prepends *Directory* to the environment search path by *Name*, unless already present. Uses semi-colon as the search-path separator on Windows operating systems, or colon everywhere else. Adds *Directory* to the start of an existing path. Makes *Directory* the first and only directory element if the search path does not yet exist.

Note that *Directory* should be an operating-system compatible search path because non-Prolog software needs to search using the included directory paths. Automatically converts incoming directory paths to operating-system compatible paths.

Note also, the environment variable *Name* is case insensitive on Windows, but not so on Unix-based operating systems.

search_path(+*Name:atom*, -*Directories:list(atom)*)** *[semidet]*
search_path(+*Name:atom*, +*Directories:list(atom)*)** *[det]*

Only fails if the environment does **not** contain the given search-path variable. Does not fail if the variable does **not** identify a proper separator-delimited variable.

search_path_separator(?*Separator:atom*)** *[semidet]*

Separator used for search paths: semi-colon on the Microsoft Windows operating system; colon elsewhere.

Chapter 39

library(os/windows): Microsoft Windows Operating System

By design, the following extensions for Windows avoid underscores in order not to clash with existing standard paths, e.g. `app_path` which Prolog defines by default.

```
userprofile
onedrive
onedrivecommercial
onedrivepersonal
programfiles
temp
documents
savedgames
appdata
applocal
localprograms
```

Chapter 40

library(paxos/http_handlers): Paxos HTTP Handlers

These handlers spool up a JSON-based HTTP interface to the Paxos predicates, namely

- `paxos_property/1` as JSON object on GET at /paxos/properties,
- `paxos_get/2` as arbitrary JSON on GET at /paxos/Key and
- `paxos_set/2` as arbitrary JSON on POST at /paxos/Key

Take the example below. Uses `http_server/1` to start a HTTP server on some given port.

```
?- [library(http/http_server), library(http/http_client)].
true.

?- http_server([port(8080)]).
% Started server at http://localhost:8080/
true.

?- http_get('http://localhost:8080/paxos/properties', A, []).
A = json([node=0, quorum=1, failed=0]).
```

Getting and setting using JSON encoding works as follows.

```
?- http_get('http://localhost:8080/paxos/hello', A, [status_code(B)]).
A = '',
B = 204.

?- http_post('http://localhost:8080/paxos/hello', json(world), A, []).
A = @true.

?- http_get('http://localhost:8080/paxos/hello', A, [status_code(B)]).
A = world,
B = 200.
```

Note that the initial GET fails. It replies with the empty atom since no content exists. Predicate `paxos_get/2` is semi-deterministic; it can fail. Empty atom is not

valid Prolog-encoding for JSON. Status code of 204 indicates no content. The Paxos ledger does not contain data for that key.

Thereafter, POST writes a string value for the key and a repeated GET attempt now answers the new consensus data. Status code 200 indicates a successful ledger concensus.

40.1 Serialisation

Serialises unknowns. Paxos ledgers may contain non-JSON compatible data. Anything that does not correctly serialise as JSON becomes an atomicly rendered Prolog term. Take a consensus value of term `a(1)` for example; GET requests see `"a(1)"` as a rendered Prolog string. The ledger comprises Prolog terms, fundamentally, rather than JSON-encoded strings.

Setting a Paxos value reads JSON from the POST request body. It can be any valid JSON value including atomic values as well as objects and arrays.

Chapter 41

library(paxos/udp_broadcast): Paxos on UDP

Sets up Paxos over UDP broadcast on port 20005. Hooks up Paxos messaging to UDP broadcast bridging using the `paxos` scope.

Initialisation order affects success. First initialises UDP broadcasting then initialises Paxos. The result is two additional threads: the UDP inbound proxy and the Paxos replicator.

You can override the UDP host, port and broadcast scope. Load settings first if you want to override using file-based settings. Back-up defaults derive from the environment and finally fall on hard-wired values of 0.0.0.0, port 20005 via `paxos` scope. You can also override the automatic Paxos node ordinal; it defaults to -1 meaning automatic discovering of unique node number. Numbers start at 0 and increase by one, translating to binary power indices for the quorum bit mask.

Note that environment defaults require upper-case variable names for Linux. Variable names match case-sensitively on Unix platforms.

41.1 Docker Stack

For Docker in production mode, your nodes want to interact using the UDP broadcast port. This port is not automatically available unless you publish it. See example snippet below. The `ports` setting lists port 20005 for UDP broadcasts across the stack.

```
version: "3"

services:

  my-service:
    image: my/image
    ports:
      - 20005:20005/udp
      - 8080:8080/tcp
```

Chapter 42

library(print/(table))

print_table(:*Goal*)** *[det]*
print_table(:*Goal*, +*Variables:list*)** *[det]*

Prints all the variables within the given non-deterministic *Goal* term formatted as a table of centre-padded columns to `current_output`. One *Goal* solution becomes one line of text. Solutions to free variables become printed cells.

Makes an important assumption: that codes equate to character columns; one code, one column. This will be true for most languages on a teletype like terminal. Ignores any exceptions by design.

```
?- print_table(user:prolog_file_type(_, _)).
+------+----------+
|  pl  |  prolog  |
|prolog|  prolog  |
| qlf  |  prolog  |
| qlf  |   qlf    |
| dll  |executable|
+------+----------+
```

Chapter 43

library(proc/loadavg)

loadavg(*-Avg1, -Avg5, -Avg15, -RunnablesRatio, -LastPID***) //** *[semidet]*
Parses the Linux `/proc/loadavg` process pseudo-file. One space separates all fields except the runnable processes and total processes, a forward slash separates these two figures.

Load-average statistics comprise: three floating point numbers, one integer ratio and one process identifier.

- Load average for last minute
- Load average for last five minutes
- Load average for last 15 minutes
- Number of currently-runnable processes, meaning either actually running or ready to run
- Total number of processes
- Last created process identifier

It follows logically that runnable processes is always less than or equal to total processes.

One space separates all fields except the runnable processes and total processes, a forward slash separates these two figures. The implementation applies this requirement explicitly. The grammar fails if more than one space exists, or if finds the terminating newline missing. This approach allows you to reverse the grammar to generate the load-average codes from the load-average figures.

loadavg(*-Avg1, -Avg5, -Avg15, -RunnablesRatio, -LastPID***)** *[det]*
Captures and parses the current processor load average statistics on Linux systems. Does **not** work on Windows systems.

throws `existence_error(source_sink, '/proc/loadavg')` on Windows, or other operating systems that do not have a `proc` subsystem.

Chapter 44

library(random/temporary)

random_temporary_module(-*M:atom*)** *[nondet]*

Finds a module that does not exist. Makes it exist. The new module has a module class of temporary. Operates non-deterministically by continuously generating a newly unique temporary module. Surround with `once/1` when generating just a single module.

Utilises the `uuid/1` predicate which never fails; the implementation relies on that prerequisite. Nor does `uuid/1` automatically generate a randomly *unique* identifier. The implementation repeats on failure to find a module that does not already exist. If the generation of a new unique module name always fails, the predicate will continue an infinite failure-driven loop running until interrupted within the calling thread.

The predicate allows for concurrency by operating a mutex across the clauses testing for an existing module and its creation. Succeeds only for mode (-).

Chapter 45

library(read/until)

read_stream_to_codes_until_end_of_file(+*In*, -*Codes***)** *[nondet]*
read_stream_to_codes_until(+*In*, -*Codes*, +*Until***)** *[nondet]*

Reads *Codes* from a stream until it finds a specific code term, such as `end_of_file`. The predicate reads the stream until it encounters the *Until* code term, which defaults to `end_of_file`. It succeeds non-deterministically for each chunk read before reaching the *Until* code term. Use this predicate to process multiple messages or data chunks from a stream, handling each chunk separately. The *Codes* variable contains the codes read from the stream, and the predicate succeeds until it reaches the *Until* condition.

Arguments

In	The input stream to read from.
Codes	The codes read from the stream.
Until	The code term that terminates the reading.

Chapter 46

library(scasp/just_dot)

scasp_just_dot_print(+*Stream*, +*Src*, +*Options***)** *[det]*

Reads a JSON file from *Src*, which is expected to be in the format produced by the `s(CASP)` solver, and prints a DOT representation of the justification graph to the specified *Stream*. The *Options* parameter allows customisation of the output, such as indentation size, graph direction, background colour, node attributes, edge attributes, and nodes to elide.

The JSON source should contain a dictionary with the following structure, simplified for clarity:

```
{
    "solver": {...},
    "query": {...},
    "answers": [
        {
            "bindings": {...},
            "model": [{"truth": ..., "value": {...}}],
            "tree": {
                "node": {"value": {...}},
                "children": [
                    {
                        "node": {"value": {...}},
                        "children": [...]
                    },
                    ...
                ]
            }
        },
        ...
    ]
}
```

The `answers` field is a list of answers, each containing bindings, a model, and a tree structure. The `tree` field represents the justification tree, where each node has a value and may have children, forming a hierarchical structure of implications.

The output is a DOT graph representation of the justification tree, where each node corresponds to a term in the justification, and edges represent implications between nodes. The graph is directed, with arrows indicating the direction of implications from one node to another.

The output is formatted as a DOT graph, which can be visualised using graph visualisation tools like Graphviz. The output can be customised using the *Options* parameter, which allows for setting various attributes of the graph, such as:

- `tab(Width)`: Specifies the indentation width for the output.
- `rankdir(Direction)`: Sets the direction of the graph layout, e.g. 'LR' for left-to-right.
- `bgcolor(Color)`: Sets the background colour of the graph.
- `node(Attributes)`: Specifies attributes for the nodes in the graph.
- `edge(Attributes)`: Specifies attributes for the edges in the graph.
- `elides(Nodes)`: A list of nodes to elide in the graph, meaning they will not be displayed.

This predicate is useful for visualising the justification structure of `s(CASP)` queries, making it easier to understand the relationships between different terms and their implications in the context of logic programming and answer set programming.

Chapter 47

library(swi/atoms)

restyle_identifier_ex(+*Style*, +*Text*, ?*Atom***)** *[semidet]*

Restyles *Text* to *Atom*. Predicate `restyle_identifier/3` fails for incoming text with leading underscore. Standard `atom:restyle_identifier/3` fails for '_' because underscore fails for `atom_codes('_', [Code])`, `code_type(Code, prolog_symbol)`. Underscore (code 95) is a Prolog variable start and identifier continuation symbol, not a Prolog symbol.

Strips any leading underscore or underscores. Succeeds only for text, including codes, but does not throw.

Arguments

Text	string, atom or codes.
Atom	restyled.

prefix_atom_suffix(?*Prefix*, ?*Atom0*, ?*Suffix*, ?*Atom***)** *[nondet]*

Non-deterministically unifies *Prefix*, *Atom0* and *Suffix* with *Atom*. Applies two `atom_concat/3` predicates in succession. Unifies from prefix to suffix for modes (?, ?, ?, -) else backwards from suffix to prefix. Empty atom is a valid atom and counts as a *Prefix*, *Suffix* or any other argument if unbound.

Chapter 48

library(swi/codes)

split_lines(*?Codes, ?Lines:list(list)***)** *[semidet]*

Splits *Codes* into *Lines* of codes, or vice versa. *Lines* split by newlines. The last line does not require newline termination. The reverse unification however always appends a trailing newline to the last line.

Chapter 49

library(swi/compounds)

flatten_slashes(+*Components0:compound, ?Components:compound*) *[semidet]*

Flattens slash-delimited components. *Components0* unifies flatly with *Components* using `mode(+, ?)`. Fails if *Components* do not match the incoming *Components0* correctly with the same number of slashes.

Consecutive slash-delimited compound terms decompose in Prolog as nested slash-functors. Compound `a/b/c` decomposes to `/(a/b, c)` for example. Subterm `a/b` decomposes to nested `/(a, b)`. The predicate converts any `/(a, b/c)` to `/(a/b, c)` so that the shorthand flattens from `a/(b/c)` to `a/b/c`.

Note that Prolog variables match partially-bound compounds; `A` matches `A/(B/C)`. The first argument must therefore be fully ground in order to avoid infinite recursion.

> **To be done** Enhance the predicate modes to allow variable components such as A/B/C; mode (?, ?).

append_path(*?Left, ?Right, ?LeftAndRight***)** *[semidet]*

LeftAndRight appends *Left* path to *Right* path. Paths in this context amount to any slash-separated terms, including atoms and compounds. Paths can include variables. Use this predicate to split or join arbitrary paths. The solutions associate to the left by preference and collate at *Left*, even though the slash operator associates to the right. Hence `append_path(A, B/5, 1/2/3/4/5)` gives one solution of A = 1/2/3 and B = 4.

There is an implementation subtlety. Only find the *Right* hand key if the argument is really a compound, not just unifies with a slash compound since Path/Component unifies with any unbound variable.

Chapter 50

library(swi/dicts): SWI-Prolog dictionary extensions

This module provides extensions to the SWI-Prolog dictionary implementation. It includes predicates for merging dictionaries, putting values into dictionaries with custom merge behavior, and handling dictionary members and leaves in a more flexible way. It also includes predicates for creating dictionaries from lists and converting dictionaries to compounds.

50.0.1 Non-deterministic ‘dict_member(?Dict, ?Member)‘

This predicate offers an alternative approach to dictionary iteration in Prolog. It makes a dictionary expose its leaves as a list exposes its elements, one by one non-deterministically. It does not unify with non-leaves, as for empty dictionaries.

```
?- dict_member(a{b:c{d:e{f:g{h:i{j:999}}}}}, Key-Value).
Key = a^b/c^d/e^f/g^h/i^j,
Value = 999.

?- dict_member(Dict, a^b/c^d/e^f/g^h/i^j-999).
Dict = a{b:c{d:e{f:g{h:i{j:999}}}}}.
```

put_dict(+*Key*, +*Dict0:dict*, +*OnNotEmpty:callable*, +*Value*, -*Dict:dict***)** *[det]*

Updates dictionary pair calling for merge if not empty. Updates *Dict0* to *Dict* with *Key-Value*, combining *Value* with any existing value by calling *OnNotEmpty*/3. The callable can merge its first two arguments in some way, or replace the first with the second, or even reject the second.

The implementation puts *Key* and *Value* in *Dict0*, unifying the result at *Dict*. However, if the dictionary *Dict0* already contains another value for the indicated *Key* then it invokes *OnNotEmpty* with the original Value0 and the replacement *Value*, finally putting the combined or selected *Value_* in the dictionary for the *Key*.

merge_dict(+*Dict0:dict, +Dict1:dict, -Dict:dict*) *[semidet]*

Merges multiple pairs from a dictionary *Dict1*, into dictionary *Dict0*, unifying the results at *Dict*. Iterates the pairs for the *Dict1* dictionary, using them to recursively update *Dict0* key-by-key. Discards the tag from *Dict1*; *Dict* carries the same tag as *Dict0*.

Merges non-dictionaries according to type. Appends lists when the value in a key-value pair has list type. Only replaces existing values with incoming values when the leaf is not a dictionary, and neither existing nor incoming is a list.

Note the argument order. The first argument specifies the base dictionary starting point. The second argument merges into the first. The resulting merge unifies at the third argument. The order only matters if keys collide. Pairs from *Dict1* replace key-matching pairs in *Dict0*.

Merging does not replace the original dictionary tag. This includes an unbound tag. The tag of *Dict0* remains unchanged after merge.

merge_pair(+*Dict0:dict, +Pair:pair, -Dict:dict*) *[det]*

Merges *Pair* with dictionary. Merges a key-value *Pair* into dictionary *Dict0*, unifying the results at *Dict*.

Private predicate `merge_dict_/3` is the value merging predicate; given the original Value0 and the incoming Value, it merges the two values at Value_.

merge_dicts(+*Dicts:list(dict), -Dict:dict*) *[semidet]*

Merges one or more dictionaries. You cannot merge an empty list of dictionaries. Fails in such cases. It does **not** unify *Dict* with a tagless empty dictionary. The implementation merges two consecutive dictionaries before tail recursion until eventually one remains.

Merging ignores tags.

dict_member(?*Dict:dict, ?Member*) *[nondet]*

Unifies with members of dictionary. Unifies *Member* with all dictionary members, where *Member* is any non-dictionary leaf, including list elements, or empty leaf dictionary.

Keys become tagged keys of the form `Tag^Key`. The caret operator neatly fits by operator precedence in-between the pair operator (`-`) and the sub-key slash delimiter (`/`). Nested keys become nested slash-functor binary compounds of the form `TaggedKeys/TaggedKey`. So for example, the compound `Tag^Key-Value` translates to Tag{Key:Value} in dictionary form. `Tag^Key-Value` decomposes term-wise as `[-, Tag^Key, Value]`. Note that tagged keys, including super-sub tagged keys, take precedence within the term.

This is a non-standard approach to dictionary unification. It turns nested sub-dictionary hierarchies into flatten pair-lists of tagged-key paths and their leaf values.

dict_leaf(-*Dict, +Pair*) *[semidet]*
dict_leaf(+*Dict, -Pair*) *[nondet]*

Unifies *Dict* with its leaf nodes non-deterministically. Each *Pair* is either

an atom for root-level keys, or a compound for nested-dictionary keys. *Pair* thereby represents a nested key path Leaf with its corresponding Value.

Fails for integer keys because integers cannot serve as functors. Does not attempt to map integer keys to an atom, since this will create a reverse conversion disambiguation issue. This **does** work for nested integer leaf keys, e.g. `a(1)`, provided that the integer key does not translate to a functor.

Arguments

Dict	is either a dictionary or a list of key-value pairs whose syntax conforms to valid dictionary data.

dict_pair(+*Dict, -Pair***)** *[nondet]*
dict_pair(-*Dict, +Pair***)** *[det]*

Finds all dictionary pairs non-deterministically and recursively where each pair is a Path-Value. Path is a slash-delimited dictionary key path. Note, the search fails for dictionary leaves; succeeds only for non-dictionaries. Fails therefore for empty dictionaries or dictionaries of empty sub-dictionaries.

findall_dict(?*Tag, ?Template, :Goal, -Dicts:list(dict)***)** *[det]*

Finds all dictionary-only solutions to *Template* within *Goal*. *Tag* selects which tags to select. What happens when *Tag* is variable? In such cases, unites with the first bound tag then all subsequent matching tags.

dict_tag(+*Dict, ?Tag***)** *[semidet]*

Tags *Dict* with *Tag* if currently untagged. Fails if already tagged but not matching *Tag*, just like `is_dict/2` with a ground tag. Never mutates ground tags as a result. Additionally Tags all nested sub-dictionaries using *Tag* and the sub-key for the sub-dictionary. An underscore delimiter concatenates the tag and key.

The implementation uses atomic concatenation to merge *Tag* and the dictionary sub-keys. Note that `atomic_list_concat/3` works for non-atomic keys, including numbers and strings. Does not traverse sub-lists. Ignores sub-dictionaries where a dictionary value is a list containing dictionaries. Perhaps future versions will.

create_dict(?*Tag, +Dict0, -Dict***)** *[semidet]*

Creates a dictionary just like `dict_create/3` does but with two important differences. First, the argument order differs. *Tag* comes first to make `maplist/3` and `convlist/3` more convenient where the Goal argument includes the *Tag*. The new dictionary *Dict* comes last for the same reason. Secondly, always applies the given *Tag* to the new *Dict*, even if the incoming Data supplies one.

Creating a dictionary using standard `dict_create/3` overrides the tag argument from its Data dictionary, ignoring the *Tag* if any. For example, using `dict_create/3` for tag xyz and dictionary abc{} gives you abc{} as the outgoing dictionary. This predicate reverses this behaviour; the *Tag* argument replaces any tag in a Data dictionary.

is_key(+*Key:any***)** *[semidet]*

Succeeds for terms that can serve as keys within a dictionary. Dictionary keys

are atoms or tagged integers, otherwise known as constant values. Integers include negatives.

Arguments	
Key	successfully unites for all dictionary-key conforming terms: atomic or integral.

dict_compound(+*Dict:dict, ?Compound:compound***)** *[nondet]*

Finds all compound-folded terms within *Dict*. Unifies with all pairs within *Dict* as compounds of the form `key(Value)` where `key` matches the dictionary key converted to one-two style and lower-case.

Unfolds lists and sub-dictionaries non-deterministically. For most occasions, the non-deterministic unfolding of sub-lists results in multiple non-deterministic solutions and typically has a plural compound name. This is not a perfect solution for lists of results, since the order of the solutions defines the relations between list elements.

Dictionary keys can be atoms or integers. Converts integers to compound names using integer-to-atom translation. However, compounds for sub-dictionaries re-wrap the sub-compounds by inserting the integer key as the prefix argument of a two or more arity compound.

list_dict(*?List, ?Tag, ?Dict***)** *[semidet]*

List to *Dict* by zipping up items from *List* with integer indexed keys starting at 1. Finds only the first solution, even if multiple solutions exist.

Chapter 51

library(swi/lists)

zip(*?List1:list, ?List2:list, ?ListOfLists:list(list)***)** *[semidet]*

Zips two lists, *List1* and *List2*, where each element from the first list pairs with the same element from the second list. Alternatively unzips one list of lists into two lists.

Only succeeds if the lists and sub-lists have matching lengths.

pairs(*?Items:list, ?Pairs:list(pair)***)** *[semidet]*

Pairs up list elements, or unpairs them in (-, +) mode. *Pairs* are First-Second terms where First and Second match two consecutive *Items*. Unifies a list with its paired list.

There needs to be an even number of list elements. This requirement proceeds from the definition of pairing; it pairs the entire list including the last. The predicate fails otherwise.

indexed(*?Items:list, ?Pairs:list(pair)***)** *[semidet]*

indexed(*?List1:list, ?Index:integer, ?List2:list***)** *[semidet]*

Unifies *List1* of items with *List2* of pairs where the first pair element is an increasing integer index. *Index* has some arbitrary starting point, or defaults to 1 for one-based indexing. Unification works in all modes.

take_at_most(*+Length:integer, +List0, -List***)** *[semidet]*

List takes at most *Length* elements from *List0*. *List* for *Length* of zero is always an empty list, regardless of the incoming *List0*. *List* is always empty for an empty *List0*, regardless of *Length*. Finally, elements from *List0* unify with *List* until either *Length* elements have been seen, or until no more elements at *List0* exist.

select1(*+Indices, +List0, -List***)** *[det]*

Selects *List* elements by index from *List0*. Applies `nth1/3` to each element of *Indices*. The 1 suffix of the predicate name indicates one-based *Indices* used for selection. Mirrors `select/3` except that the predicate picks elements from a list by index rather than by element removal.

See also

- `nth1/3`
- `select/3`

select_apply1(*+Indices, :Goal, +Extra***)** *[nondet]*

Selects one-based index arguments from *Extra* and applies these extras to *Goal*.

See also `apply/2`

comb2(*?List1, ?List2***)** *[nondet]*

Unifies *List2* with all combinations of *List1*. The length of *List2* defines the number of elements in *List1* to take at one time. It follows that length of *List1* must not be less than *List2*. Fails otherwise.

See also `http://kti.ms.mff.cuni.cz/~bartak/prolog/combinatorics.html`

Chapter 52

library(swi/memfilesio): I/O on Memory Files

author Roy Ratcliffe

52.1 Bytes and octets

Both terms apply herein. Variable names reflect the subtle but essential distinction. All octets are bytes but not all bytes are octets. Byte is merely eight bits, nothing more implied, whereas octet implies important inter-byte ordering according to some big- or little-endian convention.

with_output_to_memory_file(*:Goal, +MemoryFile, +Options***)** *[det]*
Opens *MemoryFile* for writing. Calls *Goal* using `once/1`, writing to `current_output` collected in *MemoryFile* according to the encoding within *Options*. Defaults to UTF-8 encoding.

memory_file_bytes(*?MemoryFile, ?Bytes:list***)** *[det]*
Unifies *MemoryFile* with *Bytes*.

put_bytes(*+Bytes:list***)** *[det]*
Puts zero or more *Bytes* to current output.

A good reason exists for *putting bytes* rather than writing codes. The `put_byte/1` predicate throws with permission error when writing to a text stream. *Bytes* are **not** Unicode text; they have an entirely different ontology.

> **See also** Character representation manual section at `https://www.swi-prolog.org/pldoc/man?section=chars` for more details about the difference between codes, characters and bytes.

same_memory_file(*+MemoryFile1, +MemoryFile2***)** *[semidet]*
Succeeds if, and only if, two memory files compare equal by content. Comparison operates byte-by-byte and so ignores any underlying encoding.

Chapter 53

library(swi/options)

select_options(+*Options*, +*RestOptions0*, -*RestOptions*, +*Defaults***)** *[det]*

Applies multiple `select_option/4` predicate calls to a list of *Options*. Applies the list of *Options* using a list of *Defaults*. Argument terms from *Options* unify with *RestOptions0*.

Defaults are unbound if not present. The implementation selects an option's Default from the given list of *Defaults* using `select_option/4`. Option terms must have one variable. This is because `select_option/4`'s fourth argument is a single argument. It never unifies with multiple variables even though it succeeds, e.g. `select_option(a(A, B), [], Rest, 1)` unifies A with 1, leaving B unbound.

There is a naming issue. What to call the incoming list of Option arguments and the *Options* argument with which the Option terms unify? One possibility: name the *Options* argument *RestOptions0* since they represent the initial set of *RestOptions* from which *Options* select. This clashes with `select_option/4`'s naming convention since *Options* is the argument name for *RestOptions0*'s role in the option-selection process. Nevertheless, this version follows this renamed argument convention.

The predicate is useful for selecting options from a list of options, especially when the options are not known in advance or when they need to be filtered based on certain criteria.

Example:

```
?- select_options([a(A), b(B)], [a(1), b(2), c(3)], Rest, [a(0), b(0)]).
Rest = [c(3)],
A = 1,
B = 2.
```

Arguments

Options	The list of options to select from.
RestOptions0	The initial list of remaining options.
RestOptions	The remaining options after selection.
Defaults	The list of default values for options.

Chapter 54

library(swi/paxos)

paxos_quorum_nodes(-*Nodes:list(nonneg)***)** *[semidet]*

Nodes is a list of Paxos consensus nodes who are members of the quorum. Fails if Paxos not yet initialised.

Arguments

Nodes	is a list of node indices in low-to-high order.

paxos_quorum_nth1(?*Nth1:nonneg***)** *[semidet]*

Unifies *Nth1* with the *order* of this node within the quorum. Answers 1 if this node comes first in the known quorum of consensus nodes, for example.

Chapter 55

library(swi/pengines)

pengine_collect(-*Results*, +*Options***)** *[det]*
pengine_collect(?*Template*, +*Goal*, -*Results*, +*Options***)** *[det]*

Collects Prolog engine results. Repackages the collect predicate used by the Prolog engine tests. There is only one minor difference. The number of replies maps to `replies/1` in *Options*. Succeeds if not provided but unites with the integer number of replies from all engines whenever passed to *Options*. *Options* partitions into three sub-sets: next options, state options and ask options.

The implementation utilises a mutable state dictionary to pass event-loop arguments and accumulate results. So quite useful. Note also that the second *Goal* argument is **not** module sensitive. There consequently is no meta-predicate declaration for it.

The arity-2 form of pengine_collect expects that the pengine_create options have asked a query. Otherwise the collect waits indefinitely for the engines to stop.

It is possible that the engine could exit **before** the collector asks for results. Prolog engines operate asynchronously. The collect handler pre-empts failure and avoids an ask-triggered exception by only asking existing engines for results. This does not eliminate the possibility entirely. It only narrows the window of opportunity to the interval in-between checking for existence and asking.

Arguments

Results	are the result terms, a list of successful *Goal* results accumulated by appending results from all the running engines.

pengine_wait(Options**)** *[semidet]*

Waits for Prolog engines to die. It takes time to die. If alive, wait for the engines by sampling the current engine and child engines periodically. *Options* allows you to override the default number of retries (10) and the default number of retry delays (10 milliseconds). Fails if times out while waiting for engines to die; failure means that engines remain alive (else something when wrong).

The implementation makes internal assumptions about the pengines module.

It accesses the dynamic and volatile predicates `current_pengine/6` and `child/2`. The latter is thread local.

Chapter 56

library(swi/settings)

local_settings_file(-*LocalFile:atom*) *[semidet]*

Breaks the module interface by asking for the current local settings file from the settings module. The `local_file/1` dynamic predicate retains the path of the current local file based on loading. Loading a new settings file using `load_settings/1` pushes a **new** local file without replacing the old one, so that the next `save_settings/0` keeps saving to the original file.

Arguments

LocalFile	is the absolute path of the local settings file to be utilised by the next `save_settings/1` predicate call.

setting(:*Name, ?Value, :Goal*) *[semidet]*

Semi-deterministic version of `setting/2`. Succeeds only if *Value* succeeds for *Goal*; fails otherwise. Calls *Goal* with *Value*.

Take the following example where you only want the setting predicate to succeed when it does *not* match the empty atom.

```
setting(http:public_host, A, \==(''))
```

Chapter 57

library(swi/streams)

close_streams(+*Streams:list*, -*Catchers:list***)** *[det]*
Closes zero or more *Streams* while accumulating any exceptions at *Catchers*.

Chapter 58

library(swi/zip)

zip_file_info(+*File*, -*Name*, -*Attrs*, -*Zipper***)** *[nondet]*

Non-deterministically walks through the members of a zip *File*, moving the *Zipper* current member. It does *not* read the contents of the zip members, by design. You can use the *Name* argument to select a member or members before reading.

Arguments

Zipper	unifies with the open *Zipper* for reading using `zipper_codes/3` or `zipper_open_current/3`.

zipper_codes(+*Zipper*, -*Codes*, +*Options***)** *[semidet]*

Reads the current *Zipper* file as *Codes*. *Options* may be:

- `encoding(utf8)` for UTF-8 encoded text, or
- `type(binary)` for binary octets, and so on.

Chapter 59

library(with/output)

with_output_to(+*FileType, ?Spec, :Goal***)** *[semidet]*

Runs *Goal* with `current_output` pointing at a file with UTF-8 encoding. In (+, -, :) mode, creates a randomly-generated file with random new name unified at *Spec*. With *Spec* unbound, generates a random one-time name. Does **not** try to back-track in order to create a unique random name. Hence overwrites any existing file.

This is an arity-three version of `with_output_to/2`; same name, different arity. Writes the results of running *Goal* to some file given by *Spec* and *FileType*. Fails if *Spec* and *FileType* fail to specify a writable file location.

When *Spec* unbound, generates a random name. Binds the name to *Spec*.

with_output_to_pl(?*Spec, :Goal***)** *[semidet]*

Runs *Goal* with `current_output` pointing at a randomly-generated Prolog source file with UTF-8 encoding. In (+, :) mode, creates a Prolog file with name given by *Spec*.

Index