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

Chapter 1

Canny bag o' Tudor

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 currently includes:

- Docker-style random names
- Operating system-related features:
  - Search path manipulation
  - Start and stop, upping and downing apps
- SWI-Prolog extensions for dictionaries and compounds 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!

1.1 Apps

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`.

1.1.1 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.
```

1.2 SWI-Prolog extensions

This includes the following.

1.2.1 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}}}}}.
```

Chapter 2

Change Log

Uses Semantic Versioning. Always keep a change log.

2.1 [0.7.2] - 2020-07-25

2.1.1 Added

- `append_path/3`
- `dict_pair/2`
- `take_at_most/3`
- `select1/3`, `select_apply1/3`

2.2 [0.7.1] - 2020-06-14

2.2.1 Added

- indexed_pairs/{2,3}
- `list_dict/3`
- `dict_leaf/2`
- `split_lines/2`

2.2.2 Fixed

- `make/0` warnings
- Situation debugging reports WAS, NOW
- Clean up test side effects

2.3 [0.7.0] - 2020-04-10

2.3.1 Fixed

- Key restyling for `dict_compound/2`

2.4 [0.6.1] - 2020-04-09

2.4.1 Added

- `restyle_identifier_ex/3`
- `is_key/1`
- `dict_compound/2`

2.5 [0.6.0] - 2020-04-06

2.5.1 Added

- `permute_sum_of_int/2`
- `permute_list_to_grid/2`
- `dict_tag/2`
- `print_situation_history_lengths/0`
- `create_dict/3`

2.5.2 Fixed

- Code stylings

2.6 [0.5.2] - 2020-01-11

2.6.1 Added

- `close_streams/2`

2.6.2 Fixed

- Do not independently broadcast `was/2` and `now/2` for situation transitions

2.7 [0.5.1] - 2019-12-03

2.7.1 Fixed

- Situation mutator renamed `situation_apply/2`

2.8 [0.5.0] - 2019-12-03

2.8.1 Added

- Linear algebra
- Canny maths

2.8.2 Fixed

- Canny situations

2.9 [0.4.0] - 2019-10-19

2.9.1 Added

- Canny situations
- `random_temporary_module/1` predicate
- `zip/3` predicate (swi_lists)
- `print_table/1` predicate

2.9.2 Fixed

- `with_output_to/3` uses `random_name_chk/1`

2.10 [0.3.0] - 2019-09-06

2.10.1 Added

- `random_name_chk/1` versus non-deterministic `random_name/1`

2.11 [0.2.1] - 2019-09-03

2.11.1 Fixed

- Fix the fix; `dict_member/2` unifies with empty dictionary leaf nodes

2.12 [0.2.0] - 2019-09-02

2.12.1 Fixed

- Allow dictionary leaf values for `dict_member/2`

2.13 [0.1.1] - 2019-08-02

2.13.1 Added

- Missing pack maintainer, home and download links

2.14 [0.1.0] - 2019-08-02

2.14.1 Added

- Initial spike

Chapter 3

library(canny/maths)

remainder(+*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.

Chapter 4

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 5

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)` non-deterministically.

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 non-ground 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.

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*)

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.

previously(*?Previous:any*)

previously(*?Previous:any, ?When: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 6

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 7

library(canny_tudor)

Chapter 8

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 9

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 10

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 11

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.

11.1 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*.

11.2 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.

11.3 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.

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 12

library(os/apps_debugging)

Chapter 13

library(os/apps_testing)

Chapter 14

library(os/file_searches): File searches

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.

Chapter 15

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 16

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 17

library(power_shell/get_content)

Chapter 18

library(print/(table))

print_table(*:Goal*) *[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 19

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 20

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.

Chapter 21

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 22

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. Sub-term `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 23

library(swi/dicts)

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 24

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.

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

indexed_pairs(*?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`

Chapter 25

library(swi/streams)

close_streams(+*Streams:list, -Catchers:list*) *[det]*

Closes zero or more *Streams* while accumulating any exceptions at *Catchers*.

Chapter 26

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*.