FREE eBook

LEARNING

Prolog Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#prolog

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: prolog-language

It is an unofficial and free Prolog Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Prolog Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Prolog Language

Remarks

Implementations

1. SWI-Prolog (free) swi-prolog
    - Implemented in c
2. SICStus (commercial) sicstus-prolog
3. YAP (free) yap
4. GNU Prolog (free) gnu-prolog
5. XSB (free) xsb
6. B (commercial) b-prolog
7. IF (commercial)
8. Ciao (free)
9. Minerva (commercial)
10. ECLiPSe-CLP (free) eclipse-clp
11. Jekejeke Prolog (commercial)
12. Prolog IV
13. Yield Prolog (free)
    - Implemented in c#, javascript and phyton
14. Visual Prolog (commercial) visual-prolog

Examples

Installation or Setup

SWI-Prolog

Windows and Mac:

- Download SWI-Prolog at the official website
- Simply install by following the installer instructions.

Linux (PPA):

- Add the PPA `ppa:swi-prolog/stable` to your system's software sources (developers may choose for `ppa:swi-prolog/devel`) :
    - Open a terminal (Ctrl+Alt+T) and type: `sudo add-apt-repository ppa:swi-prolog/stable`
    - Afterwards, update the package information: `sudo apt-get update`

- Now install SWI-Prolog through the package manager: `sudo apt-get install swi-prolog`
- You can now start SWI-Prolog through the command-line with command `swipl`

append/3

```
append([], Bs, Bs).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).
```

`append/3` is one of the most well-known Prolog relations. It defines a relation between three arguments and is true *if* the third argument is a list that denotes the concatenation of the lists that are specified in the first and second arguments.

Notably, and as is typical for good Prolog code, `append/3` can be used in *several directions*: It can be used to:

- *append* two fully or partially instantiated lists:

  ```
  ?- A = [1, 2, 3], B=[4, 5, 6], append(A, B, Y)
  Output:
  A = [1, 2, 3],
  B = [4, 5, 6],
  Y = [1, 2, 3, 4, 5, 6].
  ```

- *check* whether the relation is true for three fully instantiated lists:

  ```
  ?- A = [1, 2, 3], B = [4, 5], C = [1, 2, 3, 4, 5, 6], append(A, B, C)
  Output:
  false
  ```

- *generate* all possible ways to append two lists to a given list:

  ```
  ?- append(A, B, [1, 2, 3, 4]).
  Output:
  A = [],
  B = [1, 2, 3, 4] ;
  A = [1],
  B = [2, 3, 4] ;
  A = [1, 2],
  B = [3, 4] ;
  A = [1, 2, 3],
  B = [4] ;
  A = [1, 2, 3, 4],
  B = [] ;
  false.
  ```

CLP(FD) Constraints

CLP(FD) constraints are provided by all serious Prolog implementations. They allow us to reason about **integers** in a pure way.

```
?- X #= 1 + 2.
X = 3.

?- 5 #= Y + 2.
Y = 3.
```

Database Programming

Prolog categorizes everything into:

- **Atoms** - Any sequence of characters that do not start with an uppercase alphabet. Eg - `a`, `b`, `okay`
- **Numbers** - There is no special syntax for numbers, no declaration is required. Eg `1`, `22`, `35.8`
- **Variables** - A string which starts with an uppercase character or underscore (_). Eg `X`, `Y`, `Abc`, `AA`
- **Complex Terms** - They are made from a *functor* and a sequence of *arguments*. Name of a complex term is always an atom, while arguments can either be atoms or variables. Eg `father(john,doe)`, `relative(a)`, `mother(X,Y)`.

A logic database contains a set of *facts* and *rules*.

A complex term with only atoms as arguments is called a fact, while a complex term with variables as arguments is called a rule.

Example of facts in Prolog:

```
father_child(fred, susan).
mother_child(hillary, joe).
```

Example of a rule in Prolog:

```
child_of(X,Y):-
    father_child(Y,X)
    ;
    mother_child(Y,X).
```

Note that the `;` here is like the `or` operator in other languages.

Prolog is a declarative language and you can read this database as follows:

> *fred is the father of susan*
>
> *hillary is the mother of joe.*
>
> *For all `X` and `Y`, `X` is a child of `Y` if `Y` is a father of `X` or `Y` is a mother of `X`.*

In fact, a finite set of facts and or rules constitutes as a logic *program*.

The use of such a program is demonstrated by doing *queries*. Queries lets you retrieve information from a logic program.

To load the database into the interpreter (assuming that you've saved the database into the directory you are running the interpreter in) you simply enter:

```
?- [nameofdatabase].
```

replacing the `nameofdatabase` with the actual file name (note that here we exclude the `.pl` extension to the filename).

Example of queries in the interpreter for the program above and the results:

```
?- child_of(susan,fred).
true

?- child_of(joe,hillary).
true

?- child_of(fred,susan).
false

?- child_of(susan,hillary).
false

?- child_of(susan,X).
X = fred

?- child_of(X,Y).
X = susan,
Y = fred ;
X = joe,
Y = hillary.
```

The queries above and their answers can be read as follows:

> *is susan a child of fred? - true*
>
> *is joe a child of hillary? - true*
>
> *is fred a child of susan? - false*
>
> *is susan a child of hillary? - false*
>
> *who is susan a child of? - fred*

This is how we program logic in Prolog. A logic program is more formally: a set of axioms, or rules, defining relations (aka predicates) between objects. An alternative way of interpreting the database above in a more formal logic way is:

> *The relation `father_child` holds between fred and susan*
>
> *The relation `mother_child` holds between hillary and joe*
>
> *For all `X` and `Y` the relation `child_of` holds between `X` and `Y` if the relation `father_child` holds between `Y` and `X`, or the relation `mother_child` holds between `Y` and `X`.*

Hello, World in the interactive interpreter

To print "Hello, World!" in the Prolog interpreter (here we are using `swipl`, the shell for SWI Prolog):

```
$ swipl
<...banner...>
?- write('Hello, World!'), nl.
```

`?-` is the system prompt: it indicates that the system is ready for the user to enter a sequence of *goals* (i.e. a *query*) that must be terminated with a `.` (full stop).

Here the query `write('Hello World!'), nl` has two goals:

- `write('Hello World!')`: `'Hello World!'` has to be displayed **and** (`,`)
- a new line (`nl`) must follow.

`write/1` (the `/1` is used to indicate that the predicate takes one argument) and `nl/0` are *built-in predicates* (the definition is provided in advance by the Prolog system). Built-in predicates provide facilities that cannot be obtained by pure Prolog definition or to save the programmer from having to define them.

The output:

> Hello, World!
>
> yes

ends with `yes` meaning that the query has succeeded. In some systems `true` is printed instead of `yes`.

Hello, World from a file

Open a new file called `hello_world.pl` and insert the following text:

```
:- initialization hello_world, halt.

hello_world :-
    write('Hello, World!'), nl.
```

The `initialization` directive specifies that the goal `hello_world, halt` should be called when the file is loaded. `halt` exits the program.

This file can then be executed by your Prolog executable. The exact flags depend on the Prolog system. If you are using SWI Prolog:

```
$ swipl -q -l hello_world.pl
```

This will produce output `Hello, World!`. The `-q` flag suppresses the banner that usually displays when you call run `swipl`. The `-l` specifies a file to load.

Read Getting started with Prolog Language online: https://riptutorial.com/prolog/topic/1038/getting-started-with-prolog-language

Chapter 2: Coding guidelines

Examples

Naming

When programming in Prolog, we must pick two kinds of names:

- names of **predicates**
- names of **variables**.

A good *predicate* name makes clear what each argument means. By convention, **underscores** are used in names to separate the description of different arguments. This is because `underscores_keep_even_longer_names_readable`, whereas `mixingTheCasesDoesNotDoThisToTheSameExtent`.

Examples of good predicates names are:

- `parent_child/2`
- `person_likes/2`
- `route_to/2`

Note that *descriptive* names are used. Imperatives are avoided. Using descriptive names is advisable because Prolog predicates can typically be used in *multiple* directions, and the name should be applicable also of all or none of the arguments are instantiated.

Mixed capitalization is more common when selecting names of *variables*. For example: `BestSolutions`, `MinElement`, `GreatestDivisor`. A common convention for naming variables that denote successive *states* is using `S0`, `S1`, `S2`, ..., `S`, where `S` represents the final state.

Indentation

There are only a few language constructs in Prolog, and several ways for indenting them are common.

No matter which style is chosen, one principle that should always be adhered to is to **never** place `(;)/2` at the *end* of a line. This is because `;` and `,` look very similar, and `,` frequently occurs at the end of a line. Therefore, clauses that use a disjunction should for example be written as:

```
(  Goal1
;  Goal2
)
```

Order of arguments

Ideally, Prolog predicates can be used in all directions. For many pure predicates, this is also actually the case. However, some predicates only work in particular *modes*, which means instantiation patterns of their arguments.

By convention, the most common argument order for such predicates is:

- **input** arguments are placed first. These arguments must be instantiated *before* the predicate is called.
- *pairs* of arguments that belong together are placed adjacently, such as `p(..., State0, State, ...)`
- intended **output** arguments are placed last. These predicates are instantiated by the predicate.

Read Coding guidelines online: https://riptutorial.com/prolog/topic/4612/coding-guidelines

Chapter 3: Constraint Logic Programming

Examples

CLP(FD)

CLP(FD) constraints (*Finite Domains*) implement arithmetic over **integers**. They are available in all serious Prolog implementations.

There are two major use cases of CLP(FD) constraints:

- Declarative integer arithmetic
- Solving combinatorial problems such as planning, scheduling and allocation tasks.

Examples:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Note that if `is/2` were to be used in the second query, an instantiation error would occur:

```
?- 3 is Y+2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

CLP(Q)

CLP(Q) implements reasoning over *rational* numbers.

Example:

```
?- { 5/6 = X/2 + 1/3 }.
X = 1.
```

CLP(H)

Prolog itself can be considered as **CLP(H)**: Constraint Logic Programming over *Herbrand terms*. With this perspective, a Prolog program posts constraints over *terms*. For example:

```
?- X = f(Y), Y = a.
X = f(a),
Y = a.
```

Read Constraint Logic Programming online: https://riptutorial.com/prolog/topic/2057/constraint-logic-programming

Chapter 4: Control structures

Examples

Disjunction (logical OR), implicit vs. explict

Prolog tries alternative clauses for a predicate in the order of appearance:

```
likes(alice, music).
likes(bob, hiking).

// Either alice likes music, or bob likes hiking will succeed.
```

The disjunction (OR) operator ; can be used to express this in one rule:

```
likes(P,Q) :-
    ( P = alice , Q = music ) ; ( P = bob , Q = hiking ).
```

Parentheses are important here for clarity. See this Question on relative precedence for conjunction , and disjunction ;.

Conjunction (logical AND)

Conjunction (logical AND) is represented by the comma , operator (among other roles).

Conjunction between clauses can appear in a query:

```
?- X = 1, Y = 2.
```

Conjunction can also appear between the subgoal clauses in the body of a rule:

```
triangleSides(X,Y,Z) :-
    X + Y > Z, X + Z > Y, Y + Z > X.
```

Cut (remove choice points)

Sometimes it is desirable to prevent Prolog from backtracking into alternative solutions. The basic tool available to the programmer to stop prolog from continuing futher in its backtrack is the cut operator. consider the following.

```
% (percent signs mean comments)
% a is the parent of b, c, and d.
parent(a,b).
parent(a,c).
parent(a,d).
```

Here the predicate `parent/2` succeeds more than once when

```
?- parent(a,X).
```

is called. To stop prolog from searching for more solutions after the first is found you would use the cut operator, like so.

```
?- parent(a,X), !.
```

This will have X equal to b (as it is the first possible solution) and look for no more solutions.

Read Control structures online: https://riptutorial.com/prolog/topic/4479/control-structures

Chapter 5: Data Structures

Examples

Lists

Lists are a special kind of *compound term*. Lists are defined inductively:

- the atom `[]` is a list, denoting the *empty list*.
- *if* `Ls` is a list, then the term `'.'(L, Ls)` is *also* a list.

There is a special syntax for denoting lists conveniently in Prolog:

1. The list `'.'(a, '.'(b, '.'(c, [])))` can also be written as `[a,b,c]`.
2. The term `'.'(L, Ls)` can also be written as `[L|Ls]`.

These notations can be combined in any way. For example, the term `[a,b|Ls]` is a list *iff* `Ls` is a list.

Creating lists

A list consisting of literals unified with the variable List:

```
?- List = [1,2,3,4].
List = [1, 2, 3, 4].
```

Building a list by consing:

```
?- Tail = [2, 3, 4], List = [1|Tail].
Tail = [2, 3, 4],
List = [1, 2, 3, 4].
```

Building a list of unknown values using the built-in `length/2`:

```
?- length(List,5).
List = [_G496, _G499, _G502, _G505, _G508].
```

Since in Prolog everything is in essence a Term, lists behave heterogeneous:

```
?- List = [1, 2>1, this, term(X), 7.3, a-A].
List = [1, 2>1, this, term(X), 7.3, a-A].
```

This means a list can also contain other lists, also called inner lists:

```
List = [[1,2],[3,[4]]].
```

Pairs

By convention, the functor `(-)/2` is often used to denote **pairs** of elements in Prolog. For example, the term `-(A, B)` denotes the pair of elements `A` and `B`. In Prolog, `(-)/2` is defined as an *infix operator*. Therefore, the term can be written equivalently as `A-B`.

Many commonly available predicates also use this syntax to denote pairs. Examples of this are `keysort/2` and `pairs_keys_values/3`.

Association lists

In all serious Prolog systems, **association lists** are available to allow faster than linear access to a collection of elements. These association lists are typically based on *balanced trees* like **AVL trees**. There is a public domain library called `library(assoc)` that ships with many Prolog systems and provides *O(log(N))* operations for inserting, fetching and changing elements to a collection.

Terms

On a very high level, Prolog only has a single data type, called **term**. In Prolog, all data is represented by Prolog terms. Terms are defined inductively:

- an **atom** is a term. Examples of atoms are: `x`, `test` and `'quotes and space'`.
- a **variable** is a term. Variables start with an uppercase letter or underscore _.
- integers and floating point numbers are terms. Examples: `42` and `42.42`.
- a **compound term** is a term, defined inductively as follows: *If* `T1`, `T2`, ..., `T_n` are terms, *then F* (`T1`,`T2`,...,`T_n`) is also a term, where *F* is called the **functor** of the compound term.

Terms with named fields using library(record)

The `[record][1]` library provides the ability to create compound terms with named fields. The directive `:- record/1 <spec>` compiles to a collection of predicates that initialize, set and get fields in the term defined by `<spec>`.

For example, we can define a `point` data structure with named fields `x` and `y`:

```
:- use_module(library(record)).

:- record point(x:integer=0,
                y:integer=0).

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

?- default_point(Point), point_x(Point, X), set_x_of_point(10, Point, Point1).
Point = point(0, 0),
X = 0,
Point1 = point(10, 0).

?- make_point([y(20)], Point).
Point = point(0, 20).

?-  is_point(X).
false.
```

```
?- is_point(point(_, _)).
false.

?- is_point(point(1, a)).
false.

?- is_point(point(1, 1)).
true.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

Read Data Structures online: https://riptutorial.com/prolog/topic/2417/data-structures

Chapter 6: Definite Clause Grammars (DCGs)

Examples

Anything at all: `... //0`

One of the most elementary DCG nonterminals is `... //0`, which can be read as "anything at all":

```
... --> [] | [_], ... .
```

It can be used to describe a list `Ls` that contains the element `E` via:

```
phrase(( ..., [E], ... ), Ls)
```

Parsing with DCGs

DCGs can be used for parsing. Best of all, *the same* DCG can often be used to both parse and *generate* lists that are being described. For example:

```
sentence --> article, subject, verb, object.

article --> [the].

subject --> [woman] | [man].

verb --> [likes] | [enjoys].

object --> [apples] | [oranges].
```

Example queries:

```
?- phrase(sentence, Ls).
Ls = [the, woman, likes, apples] ;
Ls = [the, woman, likes, oranges] ;
Ls = [the, woman, enjoys, apples] .

?- phrase(sentence, [the,man,likes,apples]).
true .
```

Extra goals

Extra goals enable to add processing to DCG clauses, for example, conditions that the elements of the list must satisfy.

The extra goals are observed between curly braces at the end of a DCG clause.

```
% DCG clause requiring an integer
int --> [X], {integer(X)}.
```

Usage:

```
?- phrase(int, [3]).
true.

?- phrase(int, [a]).
false.
```

Extra arguments

The extra arguments add results to predicates of a DCG clause, by decorating the derivation tree. For example, it's possible to create a algebraic grammar that computes the value at the end.

Given a grammar that supports the operation addition:

```
% Extra arguments are passed between parenthesis after the name of the DCG clauses.
exp(C) --> int(A), [+], exp(B), {plus(A, B, C)}.
exp(X) --> int(X).
int(X) --> [X], {integer(X)}.
```

The result of this grammar can be validated and queried:

```
?- phrase(exp(X), [1,+,2,+,3]).
X = 6 ;
```

Read Definite Clause Grammars (DCGs) online: https://riptutorial.com/prolog/topic/2426/definite-clause-grammars--dcgs-

Chapter 7: Derivation trees

Examples

Proof tree

The proof tree (also search tree or derivation tree) is a tree that shows the execution of a Prolog program. This tree helps visualise the chronological backtracking process present in Prolog. The root of the tree represents the initial query and branches are created when choice points occur. Every node in the tree thus represents a goal. Branches only become leafs when either true/false was proven for the required (set of) goal(s) and search in Prolog is performed in a **left-to-right depth-first** fashion.

Consider following example:

```
% Facts
father_child(paul,chris).        % Paul is the father of Chris and Ellen
father_child(paul,ellen).
mother_child(ellen,angie).       % Ellen is the mother of Angie and Peter
mother_child(ellen,peter).


% Rules
grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    father_child(Z,Y).

grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    mother_child(Z,Y).
```

When we now query:

```
?- grandfather_grandchild(paul,peter).
```

following proof tree visualises the depth-first search process:

```
                                  ?- grandfather_grandchild(paul,peter).
                                      /                                \
                                     /                                  \
  ?- father_child(paul,Z1),father_child(Z1,peter).            ?-
father_child(paul,Z2),mother_child(Z2,peter).
             /                    \                                  /
\
      {Z1=chris}             {Z1=ellen}                        {Z2=chris}
{Z2=ellen}
           /                       \                               /
\
?- father_child(chris,peter).  ?- father_child(ellen,peter).  ?- mother_child(chris,peter). ?-
mother_child(ellen,peter).
         |                           |                             |
/             \
```

```
       fail                      fail                           fail
fail(*)          success
```

(*) fails for `mother_child(ellen,angie)` where 'angie' fails to match 'peter'

Read Derivation trees online: https://riptutorial.com/prolog/topic/3097/derivation-trees

Chapter 8: Difference Lists

Introduction

Difference Lists in Prolog denotes the concept to know the structure of a list *up to a point*. The remaining of the list can be left unbound until the complete evaluation of a predicate. A list where its end is unknown is referred as an *open list*, ended by a *hole*. This technique is especially useful to validate complex syntaxes or grammars.

The well-known Definite Clause Grammars (DCG) is using Difference Lists to operate under the hood.

Examples

Basic usage

Let's consider the predicate `sumDif/2`, verified if the structure of a list matches several constraints. The first term represents the list to analyze and the second term another list that holds the part of the first list that is unknown to our constraints.

For the demonstration, `sumDif/2` recognizes an arithmetic expression to sum *n* integers.

```
sumDif([X, +|OpenList], Hole) :-
    integer(X),
    sumDif(OpenList, Hole).
```

We know the first element of the list to validate is an integer, here illustrated by `X`, followed by the symbol of the addition (`+`). The remaining of the list that still needs to be processed later on (`OpenList`) is left unvalidated at that level. `Hole` represents the part of the list we *don't need* to validate.

Let's give another definition of the predicate `sumDif/2` to complete the validation of the arithmetic expression:

```
sumDif([X|Hole], Hole) :-
    integer(X).
```

We expect an integer called `X` directly at the start the open list. Interestingly, the remaining of the list `Hole` is left unknown and that's the whole purpose of the Difference Lists: the structure of the list is known up to a point.

Finally, the missing piece comes when a list is evaluated:

```
?- sumDif([1,+,2,+,3], []).
true
```

This is when the predicate is used that the end of the list is mentioned, here `[]`, indicates the list does not contain additional elements.

Evaluate an arithmetic expression

Let's define a grammar enabling us to perform additions, multiplications with the usage of parenthesis. To add more value to this example, we are going to compute the result of the arithmetic expression. Summary of the grammar:

> expression → times
> expression → times '+' expression
> times → element
> times → element '*' times
> element → "integer"
> element → '(' expression ')'

All the predicates have an arity of 3, because they need to open list, the hole and the value of the arithmetic expression.

```
expression(Value, OpenList, FinalHole) :-
    times(Value, OpenList, FinalHole).

expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).

times(Value, OpenList, FinalHole) :-
    element(Value, OpenList, FinalHole).

times(TimesValue, OpenList, FinalHole) :-
    element(Value1, OpenList, ['*'|Hole1]),
    times(Value2, Hole1, FinalHole),
    TimesValue is Value1 * Value2.

element(Value, [Value|FinalHole], FinalHole) :-
    integer(Value).

element(Value, ['('|OpenList], FinalHole) :-
    expression(Value, OpenList, [')'|FinalHole]).
```

To properly explain the principle of *holes* and how the value is computed, let's take the second clause `expression`:

```
expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).
```

The open list is denoted by the predicate `OpenList`. The first element to validate is *what comes before the addition symbol (+)*. When the first element is validated, it's directly followed by the addition symbol and by the continuation of the list, called `Hole1`. We know that `Hole1` is the next

element to validate and can be another `expression`, hence `Hole1` is then the term given to the predicate `expression`.

The value is always represented in the first term. In this clause, it's defined by the sum of the `Value1` (everything before the addition symbol) and `Value2` (everything after the addition symbol).

Finally, the an expression can be evaluated.

```
?- expression(V, [1,+,3,*,'(',5,+,5,')'], []).
V = 31
```

Read Difference Lists online: https://riptutorial.com/prolog/topic/9414/difference-lists

Chapter 9: Error handling and exceptions

Examples

Instantiation errors

An **instantiation** error is thrown if an argument is not sufficiently *instantiated*.

Critically, an instantiation error *cannot* be replaced by *silent failure*: Failing in such cases would mean that there is **no solution**, whereas an instantiation error means that an *instance* of the argument may participate in a solution.

This is in contrast to—for example—**domain error**, which can be replaced by silent failure without changing the declarative meaning of a program.

General points about error handling

Prolog features **exceptions**, which are part of the Prolog ISO standard.

An exception can be thrown with `throw/1`, and caught with `catch/3`.

The ISO standard defines many cases in which errors must or may be thrown. The standardized exceptions are all of the form `error(E,_)`, where `E` indicates the error. Examples are `instantiation_error`, `domain_error` and `type_error`, which see.

An important predicate in connection with exceptions is `setup_call_cleanup/3`, which see.

Cleaning up after exceptions

The predicate `setup_call_cleanup/3`, which is currently being considered for inclusion in the Prolog ISO standard and provided by an increasing number of implementations, lets us ensure that resources are correctly freed after an exception is thrown.

A typical invocation is:

```
setup_call_cleanup(open(File, Mode, Stream), process_file(File), close(Stream))
```

Note that an exception or interrupt may even occur immediately after `open/3` is called in this case. For this reason, the `Setup` phase is performed *atomically*. In Prolog systems that only provide `call_cleanup/2`, this is much harder to express.

Type and domain errors

A **type error** occurs if an argument is not of the expected *type*. Examples of types are:

- `integer`
- `atom`

- `list`.

If the predicate is of the expected type, but outside the expected *domain*, then a **domain error** is raised.

For example, a domain error is admissible if an integer between 0 and 15 is expected, but the argument is the integer 20.

Declaratively, a type or domain error is equivalent to *silent failure*, since no instantiation can make a predicate whose argument is of the wrong type or domain succeed.

Read Error handling and exceptions online: https://riptutorial.com/prolog/topic/7114/error-handling-and-exceptions

Chapter 10: Extra-logical Predicates

Examples

Predicates with side effects

Predicates that produce **side effects** leave the realm of pure logic. These are for example:

- `writeq/1`
- `read/1`
- `format/2`

Side effects are phenomena that cannot be reasoned about within the program. For example, deletion of a file or output on the system terminal.

Meta-logical predicates

Predicates that reason about *instantiations* are called **meta-logical**. Examples are:

- `var/1`
- `ground/1`
- `integer/1`

These predicates are outside the realm of pure monotonic logic programs, because they break properties like *commutativity* of conjunction.

Other predicates that are meta-logical include:

- `arg/3`
- `functor/3`
- `(=..)/2`

These predicates could *in principle* be modeled within first-order logic, but require an infinite number of clauses.

All-solutions predicates

Predicates that reason about *all solutions* are extra-logical. These are for example:

- `setof/3`
- `findall/3`
- `bagof/3`

!/0 and related predicates

Predicates that impede or prohibit a **declarative** reading of Prolog programs are extra-logical. Examples of such predicates are:

- `!/0`

- `(->)/2` and if-then-else
- `(\+)/1`

These predicates can only be understood procedurally, by taking into account the actual control flow of the interpreter, and as such are beyond the realm of pure logic.

Read Extra-logical Predicates online: https://riptutorial.com/prolog/topic/2282/extra-logical-predicates

Chapter 11: Higher-Order Programming

Examples

call/N predicates

The `call/N` family of predicates can call arbitrary Prolog goals at run time:

```
?- G=true, call(G).
true.

?- G=(true,false), call(G).
false.
```

maplist/[2,3]

`maplist/2` and `maplist/3` are higher-order predicates, which allow the definition of a predicate to be lifted about a single element to *lists* of such elements. These predicates can be defined using `call/2` and `call/3` as building blocks and ship with many Prolog systems.

For example:

```
?- maplist(dif(a), [X,Y,Z]).
dif(X, a),
dif(Y, a),
dif(Z, a).
```

Meta-call

In Prolog, the so-called **meta-call** is a built-in language feature. All Prolog code is represented by Prolog *terms*, allowing goals to be constructed dynamically and be used like other goals without additional predicates:

```
?- Goal = dif(X, Y), Goal.
dif(X, Y).
```

Using this mechanism, other higher-order predicates can be defined in Prolog itself.

foldl/4

A *fold* (from the left) is a higher-order relation between:

- a predicate with 3 arguments
- a list of elements
- an initial state
- a final state, which is the result of applying the predicate to successive elements while carrying through intermediate states.

For example: Use `foldl/4` to express the *sum* of all elements in a list, using a predicate as a building block to define the sum of *two* elements:

```
?- foldl(plus, [2,3,4], 0, S).
S = 9.
```

Call a list of goals

To call a list of goals as if it were a conjunction of goals, combine the higher-order predicates call/1 and maplist/2:

```
?- Gs = [X = a, Y = b], maplist(call, Gs).
Gs = [a=a, b=b],
X = a,
Y = b.
```

Read Higher-Order Programming online: https://riptutorial.com/prolog/topic/2420/higher-order-programming

Chapter 12: Logical Purity

Examples

dif/2

The predicate `dif/2` is a **pure** predicate: It can be used in all directions and with all instantiation patterns, *always* meaning that its two arguments are *different*.

CLP(FD) constraints

CLP(FD) constraints are completely pure relations. They can be used in all directions for declarative integer arithmetic:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Unification

Unification is a **pure** relation. It does not produce side effects and can be used in all directions, with either or both arguments fully or only partially instantiated.

In Prolog, unification can happen

- **explicitly**, using built-in predicates like `(=)/2` or `unify_with_occurs_check/2`
- **implicitly**, when unification is used for selecting a suitable clause.

Read Logical Purity online: https://riptutorial.com/prolog/topic/2058/logical-purity

Chapter 13: Monotonicity

Examples

Reasoning about monotonic predicates

Monotonic predicates can be debugged by applying *declarative* reasoning.

In pure Prolog, a programming mistake can lead to one or all of the following phenomena:

1. the predicate incorrectly *succeeds* in a case where it should *fail*
2. the predicate incorrectly *fails* in a case where it should *succeed*
3. the predicate unexpectedly *loops* where it should only produce a finite set of answers.

As an example, consider how we can debug case (2) by declarative reasoning: We can systematically *remove* goals of the predicate's clauses and see if the query *still* fails. In monotonic code, removing goals can at most make the resulting program *more general*. Hence, we can pinpoint errors by seeing which of the goals leads to the unexpected failure.

Examples of monotonic predicates

Examples of **monotonic** predicates are:

- **unification** with `(=)/2` or `unify_with_occurs_check/2`
- `dif/2`, expressing disequality of terms
- **CLP(FD) constraints** like `(#=)/2` and `(#>)/2`, using a monotonic execution mode.

Prolog predicates that only use monotonic goals are themselves monotonic.

Monotonic predicates allow for declarative reasoning:

1. Adding a constraint (i.e., a goal) to a query can at most *reduce*, never extend, the set of solutions.
2. Removing a goal of such predicates can at most *extend*, never reduce, the set of solutions.

Non-monotonic predicates

Here are examples of predicates that are **not** monotonic:

- meta-logical predicates like `var/1`, `integer/1` etc.
- term comparison predicates like `(@<)/2` and `(@>=)/2`
- predicates that use `!/0`, `(\+)/1` and other constructs that break monotonicity
- *all-solutions predicates* like `findall/3` and `setof/3`.

If these predicates are used, then *adding* goals can lead to more solutions, which runs counter to the important declarative property known from logic that adding constraints can at most *reduce*, never extend, the set of solutions.

As a consequence, other properties that we rely for declarative debugging and other reasoning are also broken. For example, non-monotonic predicates break the fundamental notion of **commutativity** of conjunction known from first-order logic. The following example illustrates this:

```
?- var(X), X = a.
X = a.

?- X = a, var(X).
false.
```

All-solutions predicates like `findall/3` also break monotonicity: *Adding* clauses can lead to the *failure* of goals that previously *had held*. This also runs counter to montonicity as known from first-order logic, where *adding* facts can at most *increase*, never *reduce* the set of consequences.

Monotonic alternatives for non-monotonic constructs

Here are examples of how to use **monotonic** predicates *instead* of impure, non-monotonic constructs in your programs:

- `dif/2` is meant to be used *instead* of non-monotonic constructs like `(\=)/2`
- arithmetic **constraints** (CLP(FD), CLP(Q) and others) are meant to be used *instead* of moded arithmetic predicates
- `!/0` almost always leads to non-monotonic programs and should be **avoided** entirely.
- **instantiation errors** can be raised in situations where you cannot make a sound decision at this point in time.

Combining monotonicity with efficiency

It is sometimes argued that, for the sake of efficiency, we must accept the use of non-monotonic constructs in real-world Prolog programs.

There is no evidence for this. Recent research indicates that the pure monotonic subset of Prolog may not only be sufficient to express most real-world programs, but also acceptably efficient in practice. A construct that has recently been discovered and encourages this view is `if_/3`: It combines monotonicity with a reduction of choice points. See **Indexing dif/2**.

For example, code of the form:

```
pred(L, Ls) :-
    condition(L),
    then(Ls).
pred(L, Ls) :-
    \+ condition(L),
    else(Ls).
```

Can be written with `if_/3` as:

```
pred(L, Ls) :-
    if_(condition(L),
        then(Ls),
```

```
        else(Ls)).
```

and *combines* monotonicity with determinism.

Read Monotonicity online: https://riptutorial.com/prolog/topic/3989/monotonicity

Chapter 14: Operators

Examples

Predefined operators

Predefined operators according to ISO/IEC 13211-1 and 13211-2:

Priority	Type	Operator(s)	Use
1200	xfx	`:- -->`	
1200	fx	`:- ?-`	Directive, query
1100	xfy	`;`	
1050	xfy	`->`	
1000	xfy	`','`	
900	fy	`\+`	
700	xfx	`= \=`	Term unification
700	xfx	`== \== @< @=< @> @>=`	Term comparison
700	xfx	`=..`	
700	xfx	`is =:= =\= < > =< >=`	Arithmetic evaluation and comparison
600	xfy	`:`	Module qualification
500	yfx	`+ - /\ \/`	
400	yfx	`* / div mod // rem << >>`	
200	xfx	`**`	Float power
200	xfy	`^`	Variable quantification, integer power
200	fy	`+ - \`	Arithmetic identity, negation ; bitwise complement

Many systems provide further operators as an implementation specific extension:

Priority	Type	Operator(s)	Use
1150	fx	`dynamic multifile discontiguous initialization`	Standard directives

Priority	Type	Operator(s)	Use
1150	fx	`mode public volatile block meta_predicate`	
900	fy	`spy nospy`	

Operator declaration

In Prolog, custom operators can be defined using `op/3`:

```
op(+Precedence, +Type, :Operator)
```

- Declares Operator to be an operator of a Type with a Precedence. Operator can also be a list of names in which case all elements of the list are declared to be identical operators.
- Precedence is an integer between 0 and 1200, where 0 removes the declaration.
- Type is one of: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `fy` or `fx` where `f` indicates the position of the functor and `x` and `y` indicate the positions of the arguments. `y` denotes a term with a precedence lower or equal to the precedence of the functor, whereas `x` denotes a strictly lower precedence.
    - Prefix: `fx` , `fy`
    - Infix: `xfx` (not associative), `xfy` (right associative), `yfx` (left associative)
    - Postfix: `xf` , `yf`

Example usage:

```
:- op(900, xf, is_true).

X_0 is_true :-
  X_0.
```

Example query:

```
?- dif(X, a) is_true.
dif(X, a).
```

Term ordering

Two terms may be compared via the standard ordering:

> variables @< numbers @< atoms @< strings @< structures @< lists

Notes:

- Structures compare alphabetically by functor first, then by arity and lastly by the comparison of each argument.
- Lists compare by length first, then by each element.

Order operator	Succeeds if
X @< Y	X is less than Y in the standard order
X @> Y	X is greater than Y in the standard order
X @=< Y	X is less than or equal to Y in the standard order
X @>= Y	X is greater than or equal to Y in the standard order

Example queries:

```
?- alpha @< beta.
true.

?- alpha(1) @< beta.
false.

?- alpha(X) @< alpha(1).
true.

?- alpha(X) @=< alpha(Y).
true.

?- alpha(X) @> alpha(Y).
false.

?- compound(z) @< compound(inner(a)).
true.
```

Term equality

Equality operator	Succeeds if
X = Y	X can be unified with Y
X \= Y	X cannot be unified with Y
X == Y	X and Y are identical (i.e. they unify with *no* variable bindings occurring)
X \== Y	X and Y are not identical
X =:= Y	X and Y are arithmetically equal
X =\= Y	X and Y are not arithmetically equal

Read Operators online: https://riptutorial.com/prolog/topic/2479/operators

Chapter 15: Performance

Examples

Abstract machine

For efficiency, Prolog code is typically compiled to **abstract machine code** before it is run.

Many different abstract machine architectures and variants have been proposed for efficient execution of Prolog programs. These include:

- **WAM**, the *Warren Abstract Machine*
- **TOAM**, an abstract machine used in B-Prolog.
- **ZIP**, used for example as the basis for the VM of SWI-Prolog
- **VAM**, a research architecture developped in Vienna.

Indexing

All widely used Prolog interpreters use **argument indexing** to efficiently select suitable clauses.

Users can typically rely on at least *first argument indexing*, meaning that clauses can be efficiently told apart by the functor and arity of the outermost term of the *first* argument. In calls where that argument is sufficiently instantiated, matching clauses can essentially be selected in *constant* time via hashing on that argument.

More recently, **JIT indexing** has been implemented in more systems, enabling dynamic indexing on any argument that is sufficiently instantiated when the predicate is called.

Tail call optimization

Virtually all Prolog systems implement **tail call optimization** (TCO). This means that predicate calls that are in a *tail position* can be executed in *constant* stack space if the predicate is deterministic.

Tail **recursion** optimization (TRO) is a special case of tail call optimization.

Read Performance online: https://riptutorial.com/prolog/topic/4205/performance

Chapter 16: Reasoning about data

Remarks

A new section called **Data Structures** was brought to life where explanations of certain structures + some simple example(s) of creation are provided. To keep its content concise and uncluttered, it should not contain any documentation about data manipulation.

Therefore, this section was renamed to "Reasoning about data" with as purpose the generalisation of reasoning about data in Prolog. This could include topics ranging from 'top-down inference' to 'traversal of lists', as well as many others. Because of its broad generalisation, clear subsections should be made!

Examples

Recursion

Prolog doesn't have iteration, but all iteration can be rewritten using recursion. Recursion appears when a predicate contains a goal that refers to itself. When writing such predicates in Prolog, a standard recursive pattern always has at least two parts:

- **Base (non-recursive) clause**: Typically the base-case rule(s) will represent the smallest possible example(s) of the problem that you are trying to solve - a list with no members, or just one member, or if you're working with a tree structure, it might deal with an empty tree, or a tree with just one node in it, etc. It non-recursively describes the base of the recursive process.
- **Recursive (continuing) clause**: Contains any required logic including a call to itself, continuing recursion.

As an example we shall define the well-known predicate `append/3`. Viewed declaratively, `append(L1,L2,L3)` holds when the list `L3` is the result of appending lists `L1` and `L2`. When we try to figure out the declarative meaning of a predicate, we try to describe solutions for which the predicate holds. The difficulty here lies in trying to avoid any step-by-step recurring details while still keeping in mind the procedural behaviour the predicate should exhibit.

```
% Base case
append([],L,L).

% Recursive clause
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

The base case declaratively states "any L appended to the empty list is L", note that this says nothing about L being empty – or even being a list (remember, in Prolog everything boils down to terms):

```
?- append(X,some_term(a,b),Z).
X = [],
Z = some_term(a, b).
```

For describing the recursive rule, although Prolog executes rules left-to-right, we omit the head for a second and look at the body first – reading the rule right-to-left:

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Now we say that if the body holds: “assuming that `append(L1,L2,L3)` holds”

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Then so does the head: “then so does `append([X|L1],L2,[X|L3])`”

In plain English this simply translates to:

> Assuming L3 is the concatenation of L1 and L2, then [X followed by L3] is also the concatenation of [X followed by L1] and L2.

In a practical example:

> “Assuming [1,2,3] is the concatenation of [1] and [2,3], then [a,1,2,3] is also the concatenation of [a,1] and [2,3].”

Now let's look at some queries:

It's always a good idea to initially test your predicate with the **most general query** rather than providing it with a specific scenario test case. Think of it: because of Prolog's unification, we're not required to provide test data, we just hand it free variables!

```
?- append(L1,L2,L3).
L1 = [],
L2 = L3 ;                                   % Answer #1
L1 = [_G1162],
L3 = [_G1162|L2] ;                          % Answer #2
L1 = [_G1162, _G1168],
L3 = [_G1162, _G1168|L2] ;                  % Answer #3
L1 = [_G1162, _G1168, _G1174],
L3 = [_G1162, _G1168, _G1174|L2] ;          % Answer #4
...
```

Let's replace the free variable `_G1162`-like notation with alphabetical letters to get a better overview:

```
?- append(L1,L2,L3).
L1 = [],
L2 = L3 ;                                   % Answer #1
L1 = [_A],
L3 = [_A|L2] ;                              % Answer #2
L1 = [_A, _B],
L3 = [_A, _B|L2] ;                          % Answer #3
L1 = [_A, _B, _C],
```

```
L3 = [_A, _B, _C|L2] ;                      % Answer #4
...
```

In the first answer, the base case was pattern matched and Prolog instantiated `L1` to the empty list and unified `L2` and `L3` proving that `L3` is the concatenation of the empty list and L2.

At answer #2, through chronological backtracking, the recursive clause comes into play and Prolog tries to proof that some element in the head of `L1` concatenated with `L2` is `L3` with that same element in its list head. To do so, a new free variable `_A` is unified with the head of L1 and L3 is proven to now be `[_A|L2]`.

A new recursive call is made, now with `L1 = [_A]`. Once more, Prolog tries to proof that some element placed in the head of `L1`, concatenated with `L2` is `L3` with that same element in its head. Notice that `_A` is already the head of `L1`, which perfectly matches the rule, so now, through recursion, Prolog puts `_A` in front of a new free variable and we get `L1 = [_A,_B]` and `L3 = [_A,_B|L2]`

We clearly see the recursive pattern repeating itself and can easily see that, for example, the result of the 100th step in recursion would look like:

```
L1 = [X1,X2,..,X99],
L3 = [X1,X2,..,X99|L2]
```

Note: as is typical for good Prolog code, the recursive definition of `append/3` provides us not only with the possibility of *verifying* whether a list is the concatenation of two other lists, it also *generates* all possible answers satisfying the logical relations with either fully or partially instantiated lists.

Accessing lists

Member

`member/2` has signature `member(?Elem, ?List)` and denotes `true` if `Elem` is a member of `List`. This predicate can be used to access variables in a list, where different solutions are retrieved through backtracking.

Example queries:

```
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

?- member(X,[Y]).
X = Y.

?- member(X,Y).
Y = [X|_G969] ;
Y = [_G968, X|_G972] ;
Y = [_G968, _G971, X|_G975] ;
Y = [_G968, _G971, _G974, X|_G978]
```

```
...
```

Pattern matching

When the indices you need to access are small, pattern matching can be a good solution, e.g.:

```
third([_,_,X|_], X).
fourth([_,_,_,X|_], X).
```

Read Reasoning about data online: https://riptutorial.com/prolog/topic/2005/reasoning-about-data

Chapter 17: Using Modern Prolog

Examples

Introduction

Many modern Prolog systems are in continuous development and have added new features to address classic shortcomings of the language. Unfortunately, many Prolog textbooks and even teaching courses still introduce only the outdated prolog. This topic is intended to illustrate how modern Prolog has overcome some of the problems and rather crufty syntax that appears in older Prolog and may still be being introduced.

CLP(FD) for integer arithmetic

Traditionally Prolog performed arithmetic using the `is` and `=:=` operators. However, several current Prologs offer CLP(FD) (Constraint Logic Programming over Finite Domains) as a cleaner alternative for integer arithmetic. CLP(FD) is based on storing the constraints that apply to an integer value and combining those together in memory.

CLP(FD) is an extension in most Prologs that support it, so must be loaded explicitly. Once it is loaded, the `#=` syntax can take the place of both `is` and `=:=`. For example, in SWI-Prolog:

```
?- X is 2+2.
X = 4.

?- use_module(library(clpfd)).
?- X #= 2+2.
X = 4.
```

Unlike `is`, `#=` is able to solve simple equations and unify in both directions:

```
?- 4 is 2+X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- 4 #= 2+X.
X = 2.
```

CLP(FD) provides its own generator syntax.

```
?- between(1,100,X).
X = 1;
X = 2;
X = 3...

?- X in 1..100.
X in 1..100.
```

Note that the generator does not actually run: only the range constraint is stored, ready for later

constraints to be combined with it. The generator can be forced to run (and brute force constraints) using the `label` predicate:

```
?- X in 1..100, label([X]).
X = 1;
X = 2;
X = 3..
```

Using CLP can allow some intelligent reduction of brute force cases. For example, using old-style integer arithmetic:

```
?- trace.
?- between(1,10,X), Y is X+5, Y>10.
...
Exit: (8) 6 is 1+5 ? creep
Call: (8) 6 > 10 ? creep
...
X = 6, Y = 11; ...
```

Prolog still loops through the values 1-5 even though it is mathematically provable from the given conditions that these values cannot be useful. Using CLP(FD):

```
?- X in 1..10, Y #= X+5, Y #> 10.
X is 6..10,
X+5 #= Y,
Y is 11..15.
```

CLP(FD) immediately does the maths and works out the available ranges. Adding `label([Y])` will cause X to loop only through the useful values 6..10. In this toy example, this does not increase performance because with such a small range as 1-10, the algebra processing takes as long as the loop would have done; but when a larger range of numbers are being processed this may valuably reduce computation time.

Support for CLP(FD) is variable between Prologs. The acknowledged best development of CLP(FD) is in SICStus Prolog, which is commercial and expensive. SWI-Prolog and other open Prologs often have some implementation. Visual Prolog does not include CLP(FD) in its standard library, although extension libraries for it are available.

Forall instead of failure-driven loops

Some "classic" Prolog textbooks still use the confusing and error-prone failure-driven loop syntax where a `fail` construct is used to force backtracking to apply a goal to every value of a generator. For example, to print all numbers up to a given limit:

```
fdl(X) :- between(1,X,Y), print(Y), fail.
fdl(_).
```

The vast majority of Modern Prologs no longer require this syntax, instead providing a higher order predicate to address this.

```
nicer(X) :- forall(between(1,X,Y), print(Y)).
```

Not only is this much easier to read, but if a goal that could fail was used in place of *print*, its failure would be correctly detected and passed on - whereas failures of the goals in a failure-driven loop are confused with the forced failure that drives the loop.

Visual Prolog has a custom syntactic sugar for these loops, combined with function predicates (see below):

```
vploop(X) :- foreach Y = std::fromTo(1,X) do
                 console::write(X)
             end foreach.
```

Although this looks like an imperative *for* loop, it still follows Prolog rules: in particular, each iteration of the *foreach* is its own scope.

Function-style Predicates

Traditionally in Prolog, "functions" (with one output and bound inputs) were written as regular predicates:

```
mangle(X,Y) :- Y is (X*5)+2.
```

This can create the difficulty that if a function-style predicate is called multiple times, it is necessary to "daisy chain" temporary variables.

```
multimangle(X,Y) :- mangle(X,A), mangle(A,B), mangle(B,Y).
```

In most Prologs, it is possible to avoid this by writing an alternate infix operator to use in place of `is` which expands expressions including the alternative function.

```
% Define the new infix operator
:- op(900, xfy, <-).

% Define our function in terms of the infix operator - note the cut to avoid
% the choice falling through
R <- mangle(X) :- R is (X*5)+2, !.

% To make the new operator compatible with is..
R <- X :-
    compound(X),            % If the input is a compound/function
    X =.. [OP, X2, X3],     % Deconstruct it
    R2 <- X2,               % Recurse to evaluate the arguments
    R3 <- X3,
    Expr =.. [OP, R2, R3],  % Rebuild a compound with the evaluated arguments
    R is Expr,              % And send it to is
    !.
R <- X :- R is X, !.        % If it's not a compound, just use is directly
```

We can now write:

```
multimangle(X,Y) :- X <- mangle(mangle(mangle(Y))).
```

However, some modern Prologs go further and offer a custom syntax for this type of predicate. For example, in Visual Prolog:

```
mangle(X) = Y :- Y = ((X*5)+2).
multimangle(X,Y) :- Y = mangle(mangle(mangle(X))).
```

Note that the `<-` operator and the functional-style predicate above still behave as *relations* - it is legal for them to have choice points and perform multiple unification. In the first example, we prevent this using cuts. In Visual Prolog, it is normal to use the functional syntax for relations and choice points are created in the normal way - for example, the goal `X = (std::fromTo(1,10))*10` succeeds with bindings X=10, X=20, X=30, X=40, etc.

Flow/mode declarations

When programming in Prolog it is not always possible, or desirable, to create predicates which unify for every possible combination of parameters. For example, the predicate `between(X,Y,Z)` which expresses that Z is numerically between X and Y. It is easily implemented in the cases where X, Y, and Z are all bound (either Z is between X and Y or it is not), or where X and Y are bound and Z is free (Z unifies with all numbers between X and Y, or the predicate fails if Y<X); but in other cases, such as where X and Z are bound and Y is free, there are potentially an infinite number of unifications. Although this can be implemented, it usually would not be.

Flow declaration or *mode declarations* allow an explicit description of how predicates behave when called with different combinations of bound parameters. In the case of `between`, the declaration would be:

```
%! between(+X,+Y,+Z) is semidet.
%! between(+X,+Y,-Z) is nondet.
```

Each line specifies one potential calling pattern for the predicate. Each argument is decorated with `+` to indicate cases where it is bound, or `-` to indicate cases where it is not (there are also other decorations available for more complex types such as tuples or lists that may be partially bound). The keyword after *is* indicates the behavior of the predicate in that case, and may be one of these:

- `det` if the predicate always succeeds with no choice point. For example `add(+X,+Y,-Z)` is `det` because adding two given numbers X and Y will always have exactly one answer.
- `semidet` if the predicate either succeeds or fails, with no choice point. As above, `between(+X,+Y,+Z)` is `semidet` because Z is either between X and Y or it is not.
- `multi` if the predicate always succeeds, but may have choice points (but also may not). For example, `factor(+X,-Y)` would be `multi` because a number always has at least one factor - itself - but may have more.
- `nondet` if the predicate may succeed with choice points, or fail. For example, `between(+X,+Y,-Z)` is `nondet` because there may be several possible unifications of Z to numbers between X and Y, or if Y<X then there are no numbers between them and the predicate fails.

Flow/mode declarations can also be combined with argument labeling to clarify what terms mean,

or with typing. For example, `between(+From:Int, +To:Int, +Mid:Int) is semidet.`

In pure Prologs, flow and mode declarations are optional and only used for documentation generation, but they can be extremely useful to help programmers identify the cause of instantiation errors.

In Mercury, flow and mode declarations (and types) are mandatory and are validated by the compiler. The syntax used is as above.

In Visual Prolog, flow and mode declarations and types are also mandatory and the syntax is different. The above declaration would be written as:

```
between : (int From, int To, int Mid) determ (i,i,i) nondeterm (i,i,o).
```

The meaning is the same as above, but with the differences that:

- The flow/mode declarations are separated from the type declarations (since it is assumed that flow/mode for a single predicate will not vary with type overloading);
- `i` and `o` are used for `+` and `-` and are matched with the parameters based on ordering;
- The terms used are different. `det` becomes `procedure`, `semidet` becomes `determ`, and `nondet` becomes `nondeterm` (`multi` is still `multi`).

Read Using Modern Prolog online: https://riptutorial.com/prolog/topic/5499/using-modern-prolog

Credits

S. No	Chapters	Contributors
1	Getting started with Prolog Language	coder, Community, Eyal, Limmen, manlio, mat, Nick the coder, Norman Creaney, Rajat Jain, Ruslan López Carro, SeekAndDestroy, Willem Van Onsem, Xevaquor
2	Coding guidelines	mat
3	Constraint Logic Programming	mat, SeekAndDestroy
4	Control structures	hardmath, Nick the coder
5	Data Structures	Eyal, mat, SeekAndDestroy
6	Definite Clause Grammars (DCGs)	false, mat, Will Ness, ZenLulz
7	Derivation trees	SeekAndDestroy
8	Difference Lists	ZenLulz
9	Error handling and exceptions	mat
10	Extra-logical Predicates	false, mat
11	Higher-Order Programming	4444, Eyal, mat
12	Logical Purity	mat
13	Monotonicity	manlio, mat
14	Operators	false, mat, SeekAndDestroy
15	Performance	mat
16	Reasoning about data	Daniel Lyons, manlio, mat, SeekAndDestroy
17	Using Modern Prolog	Mark Green