CHAPTER 4

# Builtins and Libraries

ProbLog supports a subset of the Prolog language for expressing models in probabilistic logic. The main difference between ProbLog's language and Prolog is that Prolog is a complete logic programming language, whereas ProbLog is a logic representation language. This means that most of the functionality of Prolog that is related to the programming part (such as control constructs and input/output) are not supported in ProbLog.

## 4.1 Supported Prolog builtins

The list of supported builtins is based on Yap Prolog. See section 6 of the Yap manual for an explanation of these predicates.

In addition: ProbLog supports `consult/1` and `use_module/1`.

### 4.1.1 Control predicates

**Supported:**

- `P, Q`
- `P; Q`
- `true/0`
- `fail/0`
- `false/0`
- `\+/1`
- `not/1`
- `call/1`
- `call/N` (for N up to 9)
- `P` (alternative to call/1)

- forall/2

**Special:**

- once/1: In ProbLog once/1 is an alias for call/1.

**Not supported:**

- !/0
- P -> Q
- P *-> Q
- repeat
- incore/1 (use call/1)
- call_with_args/N (use call/N)
- if(A,B,C) (use (A,B);(\+A,C))
- ignore/1
- abort/0
- break/0
- halt/0
- halt/1
- catch/3
- throw/1
- garbage_collect/0
- garbage_collect_atoms/0
- gc/0
- nogc/0
- grow_heap/1
- grow_stack/1

### 4.1.2 Handling Undefined Procedures

**Alternative:**

- unknown(fail) can be used

**Not supported:** all

### 4.1.3 Message Handling

**Not supported:** all

### 4.1.4 Predicates on Terms

**Supported:**

- `var/1`
- `atom/1`
- `atomic/1`
- `compound/1`
- `db_reference/1` (always fails)
- `float/1`
- `rational/1` (always fails)
- `integer/1`
- `nonvar/1`
- `number/1`
- `primitive/1`
- `simple/1`
- `callable/1`
- `ground/1`
- `arg/3`
- `functor/3`
- `T =.. L`
- `X = Y`
- `X \= Y`
- `is_list/1`
- `subsumes_term/2`

**Not supported:**

- `numbervars/3`
- `unify_with_occurs_check/2`
- `copy_term/2`
- `duplicate_term/2`
- `T1 =@= T2`
- `acyclic_term/1`

### 4.1.5 Predicates on Atoms

**Not supported:** all

**To be added:** all

### 4.1.6 Predicates on Characters

**Not supported:** all

**To be added:** all

### 4.1.7 Comparing Terms

**Supported:**

- `compare/3`
- `X == Y`
- `X \== Y`
- `X @< Y`
- `X @=< Y`
- `X @< Y`
- `X @> Y`
- `X @>= Y`
- `sort/2`
- `length/2` (both arguments unbound not allowed)

**Not supported:**

- `keysort/2`
- `predsort/2`

### 4.1.8 Arithmetic

**Supported:**

- `X`
- `-X`
- `X+Y`
- `X-Y`
- `X*Y`
- `X/Y`
- `X//Y`
- `X mod Y`
- `X rem Y` (currently same as mod)
- `X div Y`
- `exp/1`
- `log/1`
- `log10/1`

- sqrt/1
- sin/1
- cos/1
- tan/1
- asin/1
- acos/1
- atan/1
- atan/2
- sinh/1
- cosh/1
- tanh/1
- asinh/1
- acosh/1
- atanh/1
- lgamma/1
- erf/1
- erfc/1
- integer/1
- float/1
- float_fractional_part/1
- float_integer_part/1
- abs/1
- ceiling/1
- floor/1
- round/1
- sign/1
- truncate/1
- max/2
- min/2
- X ^ Y
- exp/2
- X ** Y
- X /\ Y
- X \/ Y
- X # Y
- X >< Y

- `X xor Y`
- `X << Y`
- `X >> Y`
- `\ X`
- `pi/0`
- `e/0`
- `epsilon/0`
- `inf/0`
- `nan/0`
- `X is Y`
- `X < Y`
- `X =< Y`
- `X > Y`
- `X >= Y`
- `X =:= Y`
- `X =\= Y`
- `between/3`
- `succ/2`
- `plus/3`

**Not supported:**

- `random/1`
- `rational/1`
- `rationalize/1`
- `gcd/2`
- `msb/1`
- `lsb/1`
- `popcount/1`
- `[X]`
- `cputime/0`
- `heapused/0`
- `local/0`
- `global/0`
- `random/0`
- `srandom/1`

### 4.1.9 Remaining sections

**Not supported:** all

## 4.2 ProbLog-specific builtins

- `try_call/N`: same as `call/N` but silently fail if the called predicate is undefined
- `subquery(+Goal, ?Probability)`: evaluate the Goal and return its probability
- `subquery(+Goal, ?Probability, +ListOfEvidence)`: evaluate the Goal, given the evidence, and return its Probability
- `subquery(+Goal, ?Probability, +ListOfEvidence, +Semiring, +Evaluator)`: evaluate the Goal, given the evidence, and return its Probability, using the specific semiring and evaluator (both specified using strings).
- `debugprint/N`: print messages to stderr
- `write/N`: print messages to stdout
- `writenl/N`: print messages and newline to stdout
- `nl/0`: print newline to stdout
- `error/N`: raise a UserError with some message
- `cmd_args/1`: read the list of command line arguments passed to ProbLog with the '-a' arguments
- `atom_number/2`: transfrom an atom into a number
- `nocache(Functor, Arity)`: disable caching for the predicate Functor/Arity
- `numbervars/2`:
- `numbervars/3`
- `varnumbers/2`
- `subsumes_term/2`
- `subsumes_chk/2`
- `possible/1`: Perform a deterministic query on the given term.
- `clause/2`
- `clause/3`
- `create_scope/2`
- `subquery_in_scope(+Scope, +Goal, ?Probability)`
- `subquery_in_scope(+Scope, +Goal, ?Probability, +ListOfEvidence)`
- `subquery_in_scope(+Scope, +Goal, ?Probability, +ListOfEvidence, +Semiring, +Evaluator)`
- `call_in_scope/N`
- `find_scope/2`
- `set_state/1`
- `reset_state/0`

- `check_state/1`
- `print_state/0`
- `seq/1`: Unify the variable with a sequential number. Each call generates a new sequential number.

## 4.3 Available libraries

### 4.3.1 Lists

The ProbLog lists module implements all predicates from the SWI-Prolog lists library: `memberchk/2`, `member/2`, `append/3`, `append/2`, `prefix/2`, `select/3`, `selectchk/3`, `select/4`, `selectchk/4`, `nextto/3`, `delete/3`, `nth0/3`, `nth1/3`, `nth0/4`, `nth1/4`, `last/2`, `proper_length/2`, `same_length/2`, `reverse/2`, `permutation/2`, `flatten/2`, `max_member/2`, `min_member/2`, `sum_list/2`, `max_list/2`, `min_list/2`, `numlist/3`, `is_set/1`, `list_to_set/2`, `intersection/3`, `union/3`, `subset/2`, `subtract/3`.

In addition to these, the ProbLog library provides the following:

**`select_uniform(+ID, +Values, ?Value, ?Rest)`** ...

**`select_weighted(+ID, +Weights, +Values, ?Value, ?Rest)`** ...

**`groupby(?List, ?Groups)`** ...

**`sub_list(?List, ?Before, ?Length, ?After, ?SubList)`** ...

**`enum_groups(+Groups, +Values, -Group, -GroupedValues)`** ...

**`enum_groups(+GroupValues, -Group, -GroupedValues)`** ...

**`unzip(ListAB,ListA,ListB)`** ...

**`zip(ListA,ListB,ListAB)`** ...

**`make_list(Len,Elem,List)`** ...

### 4.3.2 Apply

The ProbLog lists module implements all predicates from the SWI-Prolog apply library: `include/3`, `exclude/3`, `partition/4`, `partition/5`, `maplist/2`, `maplist/3`, `maplist/4`, `maplist/5`, `convlist/3`, `foldl/4`, `foldl/5`, `foldl/6`, `foldl/7`, `scanl/4`, `scanl/5`, `scanl/6`, `scanl/7`.

### 4.3.3 Cut

ProbLog does not support cuts (`!`). However, it does provide the cut library to help with the modeling of ordered rulesets.

This library implements a soft cut.

1. Define a set of indexed-clauses (index is first argument)

```
r(1, a, b).
r(2, a, c).
r(3, b, c).
```

2. Call the rule using cut where you should remove the first argument

```
cut(r(A, B))
```

This will evaluate the rules in order of their index (note: NOT order in the file) and only ONE rule will match (the first one that succeeds).

e.g.:

```
cut(r(A, B)) => A = a, B = b
cut(r(a, X)) => X = b
cut(r(X, c)) => X = a
cut(r(b, X)) => X = c
```

The predicate cut/2 unifies the second argument with the Index of the matching rule.

### 4.3.4 Assert

The assert module allows assert and retracting facts dynamically from the internal database.

It provides the predicates `assertz/1`, `retract/1`, `retractall/1`.

### 4.3.5 Record

The record module allows access to non-backtrackable storage in the internal database.

It provides the predicates `current_key/1`, `recorda/2`, `recorda/3`, `recordz/2`, `recordz/3`, `erase/1`, `recorded/2`, `recorded/3`, `instance/2`.

### 4.3.6 Aggregate

The `aggregate` library LDL++ style of aggregation.

This functionality requires the 'aggregate' library.

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

An aggregating clause is a clause of the form:

```
FUNCTOR(*GroupArgs, AggFunc<AggVar>) :- BODY.
```

with

- FUNCTOR: The predicate name.
- GroupArgs: (optional) list of arguments that will be used as a "group by". That is, the clause will produce a result for each distinct set
- AggFunc: An aggregation function. This can be any binary predicate that maps a list onto a term.
- AggVar: The variable over which the aggregation is computed.
- BODY: The body of the clause.

The library provides 'sum', 'avg', 'min' and 'max', but also user-defined predicates can be used.

User defined predicates have to be /2, with a list as input and some result as output. For example, the predicate proper_length/2 in lists fits this definition and can be used natively as an aggregation.

Examples

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

person(a).
person(b).
person(c).
person(d).
person(e).

salary(a, 1000).
salary(b, 1200).
salary(c, 800).
salary(d, 1100).
salary(e, 1400).

dept(a, dept_a).
dept(b, dept_a).
dept(c, dept_b).
dept(d, dept_b).
dept(e, dept_a).

% Average salary per department.
dept_salary(Dept, avg<Salary>) :- person(X), salary(X, Salary), dept(X, Dept).
query(dept_salary(Dept, Salary)).
% dept_salary(dept_a,1200.0) 1
% dept_salary(dept_b,950.0) 1

% Max salary per department.
dept_max_salary(Dept, max<Salary>) :- person(X), salary(X, Salary), dept(X, Dept).
query(dept_max_salary(Dept, Salary)).
% dept_max_salary(dept_a,1400) 1
% dept_max_salary(dept_b,1100) 1

% Average salary company-wide.
all_salary(avg<Salary>) :- person(X), salary(X, Salary), dept(X, Dept).
query(all_salary(Salary)).
% all_salary(1100.0) 1
```

These aggregates also support probabilistic data.

### 4.3.7 Collect

The `collect` library provides the => operator generalizing the operator `all/3`.

The general syntax of this operator is:

```
( CODEBLOCK ) => GroupBy / AggFunc(Arg1, Arg2, ..., ArgK)
```

with

- CODEBLOCK: A block of code parseable by Prolog
- AggFunc: An aggregation function to apply on the result of evaluating CODEBLOCK.
- Arg1, . . . , ArgK: An arbitrary number of arguments to the aggregation function.
- GroupBy: An optional expression over the aggregation function should be grouped.

In order to implement the aggregation operator, the user should define a predicate

```
collect_AggFunc(CodeBlock, GroupBy, Arg1, Arg2, ..., ArgK, Result)
```

Where standard aggregation function (e.g., the functions provided by the `aggregate` library) can be collected using the operator `aggregate/5` from the `aggregate` library through

```
collect_AggFunc(CodeBlock, GroupBy, AggVar, AggRes) :-
    aggregate(AggFunc, AggVar, GroupBy, CodeBlock, (GroupBy, AggRes)).
```

Considering predicates `cell(Row, Column, Value)` and `cell_type(Row, Column, Type)` we could use => to get the average per column of cell values representing an integer.

e.g.:

```
column_average(Column, Avg) :- (
   cell(Row, Column, Value),
   type(cell(Row, Column, 'int')
) => Column / avg(Value, Avg).
```

Where `collect_avg` can be defined using the operator `avg/2` from the `aggregate` library

```
collect_avg(CodeBlock, GroupBy, AggVar, AggRes) :-
   aggregate(avg, AggVar, GroupBy, CodeBlock, (GroupBy, AggRes)).
```

### 4.3.8 DB

The `db` library provides access to data stored in an SQLite database or a CSV-file. It provides two predicates:

**`sqlite_load(+Filename)`** This creates virtual predicates for each table in the database.

**`sqlite_csv(+Filename, +Predicate)`** This creates a new predicate for the data in the CSV file.

For a demonstration on how to use these, see this tutorial article.

### 4.3.9 Scope

In order to manage several Problog theories in one model, theories can be defined through the scope operator `:/2`. The left member of the scope is the scope name and its right member the predicate in the scope. e.g.:

```
scope(1):knowledge(1).
```

Scopes can be manipulated as set of predicates.

e.g., the union of scopes can be generated through the `;/2` operator and a whole scope can be queried through the unification of its predicates:

```
scope(1):a.
scope(2):b.
scope(3):X :- scope(1):X; scope(2):X.
query(scope(3):_).

result:
 scope(3):a:    1
 scope(3):b:    1
```

The `scope` library provides additional behaviours in scopes.

Conjunction reasoning, e.g.:

```
scope(1):a.
scope(1):b.
query(scope(1):(a,b)).

result:
 scope(1):(a, b):  1
```

Temporary union through list, e.g.:

```
scope(1):a.
scope(2):b.
query([scope(1),scope(2)]:b).

result:
 [scope(1), scope(2)]:b:   1
```

All predicates outside any scope are considered in all scopes, e.g:

```
a.
query(scope(1):a).

result:
 scope(1):a:   1
```

### 4.3.10 String

The `string` library provides predicates for string manipulation.

### 4.3.11 NLP4PLP

A library for representing and solving probability questions. See the NLP4PLP webpage for more information.