The Aleph Manual

Version 4 and above

Then the rabbi said, "Golem, you have not been completely formed, but I am about to finish you now…You will do as I will tell you." Saying these words, Rabbi Leib finished engraving the letter Aleph. Immediately the golem began to rise." From *The Golem* by Isaac Bashevis Singer with illustrations by Uri Shulevitz.

Ashwin Srinivasan

The development of Aleph owes much to the advice and research of many people. Special thanks are due to: Michael Bain, Rui Camacho, Vitor Costa, James Cussens, Ross King, Donald Michie, Stephen Moyle, Stephen Muggleton, David Page, Mark Reid, Claude Sammut, Jude Shavlik, Jan Wielemaker, and Filip Zelezny.

1 Introduction

This document provides reference information on **A** **L**earning **E**ngine for **P**roposing **H**ypotheses (Aleph). Aleph is an Inductive Logic Programming (ILP) system. This manual is not intended to be a tutorial on ILP. A good introduction to the theory, implementation and applications of ILP can be found in S.H. Muggleton and L. De Raedt (1994), *Inductive Logic Programming: Theory and Methods*, Jnl. Logic Programming, 19,20:629–679, available at `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/lpj.ps.gz`.

Aleph is intended to be a prototype for exploring ideas. Earlier incarnations (under the name P-Progol) originated in 1993 as part of a fun project undertaken by Ashwin Srinivasan and Rui Camacho at Oxford University. The main purpose was to understand ideas of inverse entailment which eventually appeared in Stephen Muggleton's 1995 paper: *Inverse Entailment and Progol*, New Gen. Comput., 13:245-286, available at `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/InvEnt.ps.gz`. Since then, the implementation has evolved to emulate some of the functionality of several other ILP systems. Some of these of relevance to Aleph are: CProgol, FOIL, FORS, Indlog, MIDOS, SRT, Tilde, and WARMR. See Chapter 4 [Other Programs], page 39 for more details on obtaining some of these programs.

1.1 How to obtain Aleph

Aleph is written in Prolog principally for use with the Yap Prolog compiler. It should also run, albeit less efficiently, with SWI Prolog. It is maintained by Ashwin Srinivasan at the University of Oxford, and can be found at:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.pl`.

If you obtain this version, and have not already done so, then subscribe to the Aleph mailing list. You can do this by e-mailing `majordomo@comlab.ox.ac.uk` with the body of the message containing the command: **subscribe aleph**. This version is free for academic use (research and teaching). If you intend to use it for commercial purposes then please contact Ashwin Srinivasan (ashwin at comlab dot ox dot ac uk).

NB: Yap is available at:

`http://yap.sourceforge.net/`

Aleph requires Yap 4.1.15 or higher, compiled with the DEPTH_LIMIT flag set to 1 (that is, include -DDEPTH_LIMIT=1 in the compiler options). Aleph 5 requires SWI Version 5.1.10 or higher.

SWI Prolog is available at:

`http://www.swi-prolog.org/`

1.2 How to use this manual

- If you are a first-time user, proceed directly to Section 1.3 [Aleph Algorithm], page 2.
- If you have mastered the naive use of Aleph then see Chapter 3 [Advanced Use], page 9 on how to get more out of this program. You may also want to look at the ⟨undefined⟩ [Concept Index], page ⟨undefined⟩.

- If you are familiar with idea of setting parameters, altering search methods, etc within Aleph, then see Chapter 5 [Notes], page 41 for ideas that have proved worthwhile in applications.
- If you are interested in what is new with this version, see Chapter 6 [Change Logs], page 49 for a change-log.

The Texinfo source file of this manual is available at:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/aleph.tex`

A "Makefile" is available for generating a variety of output formats:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/Makefile.txt`

1.3 The basic Aleph algorithm

During routine use, Aleph follows a very simple procedure that can be described in 4 steps:

1. **Select example.** Select an example to be generalised. If none exist, stop, otherwise proceed to the next step.
2. **Build most-specific-clause.** Construct the most specific clause that entails the example selected, and is within language restrictions provided. This is usually a definite clause with many literals, and is called the "bottom clause." This step is sometimes called the "saturation" step. Details of constructing the bottom clause can be found in Stephen Muggleton's 1995 paper: *Inverse Entailment and Progol*, New Gen. Comput., 13:245-286, available at `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/InvEnt.ps.gz`.
3. **Search.** Find a clause more general than the bottom clause. This is done by searching for some subset of the literals in the bottom clause that has the "best" score. Two points should be noted. First, confining the search to subsets of the bottom clause does not produce all the clauses more general than it, but is good enough for this thumbnail sketch. Second, the exact nature of the score of a clause is not really important here. This step is sometimes called the "reduction" step.
4. **Remove redundant.** The clause with the best score is added to the current theory, and all examples made redundant are removed. This step is sometimes called the "cover removal" step. Note here that the best clause may make clauses other than the examples redundant. Again, this is ignored here. Return to Step 1.

A more advanced use of Aleph (see Chapter 3 [Advanced Use], page 9) allows alteration to each of these steps. At the core of Aleph is the "reduction" step, presented above as a simple "subset-selection" algorithm. In fact, within Aleph, this is implemented by a (restricted) branch-and-bound algorithm which allows an intelligent enumeration of acceptable clauses under a range of different conditions. More on this can be found in Section 5.7 [Aleph Implementation], page 43.

2 Getting started with Aleph

2.1 Loading Aleph

Aleph code is contained in a single file, usually called `alephX.pl` (the X stands for the current version number, for example aleph4.pl refers to Version 4). To load Aleph, you will need to consult this file into your Prolog compiler, with sufficient stack and heap size (the more, the better!). Here is an example of loading Aleph into the Yap compiler, with a stack size of 5000 K bytes and heap size of 20000 K bytes:

```
yap -s5000 -h20000

[ Restoring file startup ]

yes

   ?- [aleph4].
```

Aleph requires 3 files to construct theories. The most straightforward use of Aleph would involve:

1. Construct the 3 data files called `filestem.b, filestem.f, filestem.n`. See Section 2.2 [Background Knowledge File], page 3, Section 2.3 [Positive Examples File], page 5, and Section 2.4 [Negative Examples File], page 5.
2. Read all data using the `read_all(filestem)` command. See Section 2.5 [Read Input Files], page 5.
3. Construct a theory using the `induce` command See Section 2.6 [Construct Theory], page 6.

2.2 Background knowledge file

All background knowledge for Aleph is contained in a file with a **.b** extension. Background knowledge is in the form of Prolog clauses that encode information relevant to the domain. It can also contain any directives understood by the Prolog compiler being used (for example, `:- consult(someotherfile).`). This file also contains language and search restrictions for Aleph. The most basic amongst these refer to *modes, types* and *determinations* (see Section 2.2.1 [Modes], page 3, Section 2.2.2 [Types], page 4, and Section 2.2.3 [Determinations], page 4).

2.2.1 Mode declarations

These declare the mode of call for predicates that can appear in any clause hypothesised by Aleph. They take the form:

```
mode(RecallNumber,PredicateMode).
```

where `RecallNumber` bounds the non-determinacy of a form of predicate call, and `PredicateMode` specifies a legal form for calling a predicate.

`RecallNumber` can be either (a) a number specifying the number of successful calls to the predicate; or (b) `*` specifying that the predicate has bounded non-determinacy. It is usually easiest to specify `RecallNumber` as `*`.

`PredicateMode` is a template of the form:

```
p(ModeType, ModeType,...)
```

Here are some examples of how they appear in a file:

```
:- mode(1,mem(+number,+list)).
:- mode(1,dec(+integer,-integer)).
:- mode(1,mult(+integer,+integer,-integer)).
:- mode(1,plus(+integer,+integer,-integer)).
:- mode(1,(+integer)=(#integer)).
:- mode(*,has_car(+train,-car)).
```

Each ModeType is either (a) **simple**; or (b) **structured**. A **simple** ModeType is one of: (a) **+T** specifying that when a literal with predicate symbol **p** appears in a hypothesised clause, the corresponding argument should be an "input" variable of type **T**; (b) **-T** specifying that the argument is an "output" variable of type **T**; or (c) **#T** specifying that it should be a constant of type **T**. All the examples above have simple modetypes. A **structured** ModeType is of the form **f(..)** where **f** is a function symbol, each argument of which is either a simple or structured ModeType. Here is an example containing a structured ModeType:

```
:- mode(1,mem(+number,[+number|+list])).
```

With these directives Aleph ensures that for any hypothesised clause of the form `H:- B1, B2, ..., Bc`:

1. **Input variables.** Any input variable of type **T** in a body literal Bi appears as an output variable of type **T** in a body literal that appears before Bi, or appears as an input variable of type **T** in H.
2. **Output variables.** Any output variable of type **T** in H appears as an output variable of type **T** in Bi.
3. **Constants.** Any arguments denoted by **#T** in the modes have only ground terms of type **T**.

2.2.2 Type specifications

Types have to be specified for every argument of all predicates to be used in constructing a hypothesis. This specification is done within a `mode(...,...)` statement (see Section 2.2.1 [Modes], page 3). For Aleph types are just names, and no type-checking is done. Variables of different types are treated distinctly, even if one is a sub-type of the other.

2.2.3 Determinations

Determination statements declare the predicated that can be used to construct a hypothesis. They take the form:

```
determination(TargetName/Arity,BackgroundName/Arity).
```

The first argument is the name and arity of the target predicate, that is, the predicate that will appear in the head of hypothesised clauses. The second argument is the name and arity of a predicate that can appear in the body of such clauses. Typically there will be many determination declarations for a target predicate, corresponding to the predicates thought to be relevant in constructing hypotheses. If no determinations are present Aleph does not construct any clauses. Determinations are only allowed for 1 target predicate on any given run of Aleph: if multiple target determinations occur, the first one is chosen.

Here are some examples of how they appear in a file:

```
:- determination(eastbound/1,has_car/2).
:- determination(mult/3,mult/3).
:- determination(p/1,'='/2).
```

2.3 Positive examples file

Positive examples of a concept to be learned with Aleph are written in a file with a **.f** extension. The filestem should be the same as that used for the background knowledge. The positive examples are simply ground facts. Here are some examples of how they appear in a file:

```
eastbound(east1).
eastbound(east2).
eastbound(east3).
```

Code exists for dealing with non-ground positive examples. However, this has never been tested rigorously.

2.4 Negative examples file

Negative examples of a concept to be learned with Aleph are written in a file with a **.n** extension. The filestem should be the same as that used for the background knowledge. The negative examples are simply ground facts. Here are some examples of how they appear in a file:

```
eastbound(west1).
eastbound(west1).
eastbound(west1).
```

Non-ground constraints can be a more compact way of expressing negative information. Such constraints can be specified in the background knowledge file (see Section 3.2.5 [Constraints], page 23). Aleph is capable of learning from positive examples only. This is done using a Bayesian evaluation function (see `posonly` in Section 3.2.2 [Search Function], page 21).

2.5 Read all input files

Once the `filestem.b`, `filestem.f` and `filestem.n` files are in place, they can be read into Aleph with the command:

```
read_all(filestem).
```

Finer-grain specification of the example files can be achieved by setting the `train_pos` and `train_neg` flags (see Section 3.1 [Other Settings], page 10).

2.6 Construct a theory

The basic command for selecting examples and constructing a theory is:

```
induce.
```

When issued Aleph does the four steps described earlier (see Section 1.3 [Aleph Algorithm], page 2). The result is usually a trace that lists clauses searched along with their positive and negative example coverage, like:

```
eastbound(A) :-
   has_car(A,B).
[5/5]
eastbound(A) :-
   has_car(A,B), short(B).
[5/5]
eastbound(A) :-
   has_car(A,B), open_car(B).
[5/5]
eastbound(A) :-
   has_car(A,B), shape(B,rectangle).
[5/5]
```

and the final result that looks like:

```
[theory]

[Rule 1] [Pos cover = 5 Neg cover = 0]

eastbound(A) :-
       has_car(A,B), short(B), closed(B).

[pos-neg] [5]
```

`induce` also reports the performance on the training data as a confusion matrix that looks like:

```
[Training set performance]

          Actual
       +         -
    +  5         0         5
```

```
Pred
     -  0          5          5

        5          5          10

Accuracy = 100%
```

Performance on a test data is also reported if values for the parameters `test_pos` and `test_neg` are set (see Section 3.1 [Other Settings], page 10).

The simplest use of `induce` implements a simple greedy cover-set algorithm. Aleph allows you to experiment with a number of other ways of searching for answers (see Chapter 3 [Advanced Use], page 9).

2.7 Save a theory

The final theory constructed by Aleph can be saved in a file `FileName` using the command:

```
write_rules(FileName).
```

Alternatively, the command:

```
write_rules.
```

calls `write_rules/1` with the current setting for the parameter `rulefile`.

2.8 Evaluate a theory

Besides automatic performance reporting, the theory constructed by Aleph can be evaluated on examples in any data file using the command:

```
test(File,Flag,Covered,Total)
```

`File` is the name of the data file containing the examples. `Flag` is one of `show` or `noshow` to show examples covered or otherwise. Both `File` and `Flag` have to be provided. `test/4` then returns the following numbers. `Covered` is the number of examples in the data file covered by current theory. `Total` is the total number of examples in the data file.

2.9 Some simple examples

Some simple examples of Aleph usage can be found in

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

In each sub-directory you should find Aleph input files and, usually, a typescript of Aleph running on the data provided to accomplish some task.

3 Advanced use of Aleph

Advanced use of Aleph allows modifications to each of the steps to the basic algorithm (see Section 1.3 [Aleph Algorithm], page 2):

1. **Select example.** A sample of more than 1 example can be selected (see `samplesize` in Section 3.1 [Other Settings], page 10). The best clause obtained from reducing each corresponding bottom clause is then added to the theory. Alternatively, no sampling need be performed, and every example can be saturated and reduced (see `induce` in Section 3.2 [Other Searches], page 19).
2. **Build most-specific-clause.** Bottom clauses may be constructed "lazily" or not at all (see `construct_bottom` in Section 3.1 [Other Settings], page 10). Literals in the a bottom clause may be evaluated "lazily" (see `lazy_evaluate` in Section 3.11 [Other Commands], page 34). Individual bottom clauses can be constructed and examined (see `sat` in Section 3.11 [Other Commands], page 34).
3. **Search.** The search for clauses can be altered and customised to try different search strategies, evaluation functions, and refinement operators (see Section 3.2 [Other Searches], page 19). A bottom clause can be reduced repeatedly using different search constraints (see `reduce` in Section 3.11 [Other Commands], page 34).
4. **Remove redundant.** Examples covered may be retained to give better estimates of clause scores (see `induce` in Section 3.2 [Other Searches], page 19).

There is now some software in place that allows exploration of the following:

1. **Randomised search.** The basic Aleph algorithm does a fairly standard general-to-specific search. Some variation on this is possible by the user specifying his or her own refinement operator. In other areas (satisfiability of propositional formulae, simulation of discrete events), randomised methods have proven extremely useful tools to search very large spaces. The implementation within Aleph is an adaptation of the standard randomised methods: GSAT, WSAT, RRR, and the Metropolis algorithm (a special case of simulated annealing with a fixed 'temperature') (see Section 3.3 [Randomised Search], page 24 and Section 3.2 [Other Searches], page 19).
2. **Incremental learning.** The basic Aleph algorithm is a "batch" learner in the sense that all examples and background are expected to be in place before learning commences. An incremental mode allows Aleph to acquire new examples and background information by interacting with the user (see Section 3.4 [Incremental Learning], page 26).
3. **Theory learning.** The basic Aleph algorithm constructs a "theory" one clause at a time. This is an implementation of the greedy set-cover algorithm to the problem of identifying a set of clauses. There is some empirical and theoretical work done on on ILP of sets of clauses at once: see the work of I. Bratko and H. Midelfart in *Proceedings of the Ninth International Workshop on Inductive Logic Programming (ILP'99)*, LNAI-1634. Theory learning by Aleph uses randomised search methods (see next) to search through the space of theories. It has not been tested to any significant extent (see Section 3.5 [Theory Learning], page 27).
4. **Learning trees.** The basic Aleph algorithm constructs clauses using a greedy set-covering algorithm. In some sense, this can be seen as the first-order equivalent of propositional rule-learning algorithms like Clark and Niblett's CN2. There is now a substantial body of empirical work (done by researchers in Leuven and Freiburg)

demonstrating the utility of first-order equivalents of propositional tree-learning procedures. Tree-based learning can be seen as a special case of theory learning and the implementation in Aleph uses the standard recursive-partitioning approach to construct classification, regression, class probability, or model trees (see Section 3.6 [Tree Learning], page 28).

5. **Learning constraints.** The basic Aleph algorithm constructs definite clauses normally intended to be components of a predictive model for data. Early ILP work (in the Claudien system) demonstrated the value of discovering all non-Horn constraints that hold in a database. The implementation of these ideas in Aleph uses a naive generate-and-test strategy to enumerate all constraints within the mode language provided (see Section 3.7 [Constraint Learning], page 30).
6. **Learning modes.** The basic Aleph algorithm assumes modes will be declared by the user. There has been some work (by McCreath and Sharma) on automatic extraction of mode and type information from the background knowledge provided. The implementation of these ideas in Aleph follows these ideas fairly closely (see Section 3.8 [Mode Learning], page 31).
7. **Learning features.** The basic Aleph algorithm constructs a set of rules that, along with the background knowledge, entail the positive examples. Good clauses found during the search for this set of rules can be used to construct boolean features. These can then be used techniques like maximum entropy modelling, support vector machines and so on (see Section 3.10 [Feature Construction], page 33).

These are all at very early stages of development and therefore even less reliable than the rest of the code (probably).

3.1 Setting Aleph parameters

The `set/2` predicate forms the basis for setting a number of parameter values for Aleph. Parameters are set to values using:

```
set(Parameter,Value)
```

The current value of a parameter is obtained using:

```
setting(Parameter,Value)
```

A parameter can be un-set by using:

```
noset(Parameter)
```

Meaningful `set/2` statements for Aleph are:

`set(abduce,+`*V*`)`

V is one of: `true` or `false` (default `false`). If *V* is `true` then abduction and subsequent generalisation of abduced atoms is performed within the `induce` loop. Only predicates declared to be abducible by `abducible/1` are candidates for abduction. See Section 3.9 [Abductive Learning], page 32 for more details.

`set(best,+`*V*`)`

V is a 'clause label' obtained from an earlier run. This is a list containing at least the number of positives covered, the number of negatives covered, and the length of a clause found on a previous search. Useful when performing searches iteratively.

`set(cache_clauselength,+V)`
: *V* is a positive integer (default 3). Sets an upper bound on the length of clauses whose coverages are cached for future use.

`set(caching,+V)`
: *V* is one of: `true` or `false` (default `false`). If `true` then clauses and coverage are cached for future use. Only clauses upto length set by `cache_clauselength` are stored in the cache.

`set(check_redundant,+V)`
: *V* is one of: `true` or `false` (default `false`). Specifies whether a call to `redundant/2` (see Section 3.11 [Other Commands], page 34) should be made for checking redundant literals in a clause.

`set(check_useless,+V)`
: *V* is one of: `true` or `false` (default `false`). If set to `true`, removes literals in the bottom clause that do not contribute to establishing variable chains to output variables in the positive literal, or produce output variables that are not used by any other literal in the bottom clause.

`set(classes,+V)`
: *V* is a list of classes to be predicted by the tree learner (see Section 3.6 [Tree Learning], page 28).

`set(clauselength,+V)`
: *V* is a positive integer (default 4). Sets upper bound on number of literals in an acceptable clause.

`set(clauselength_distribution,+V)`
: *V* is a list of the form [p1-1,p2-2,...] where "pi" represents the probability of drawing a clause with "i" literals. Used by randomised search methods See Section 3.3 [Randomised Search], page 24.

`set(clauses,+V)`
: *V* is a positive integer. Sets upper bound on the number of clauses in a theory when performing theory-level search (see Section 3.5 [Theory Learning], page 27).

`set(condition,+V)`
: *V* is one of: `true` or `false` (default `false`). If `true` then randomly generated examples are obtained after conditioning the stochastic generator with the positive examples.

`set(confidence,+V)`
: *V* is a floating point number in the interval (0.0,1.0) (default 0.95). Determines the confidence for rule-pruning by the tree learner (see Section 3.6 [Tree Learning], page 28).

`set(construct_bottom,+V)`
: *V* is one of: `saturation`, `reduction` or `false` (default `saturation`). Specifies the stage at which the bottom clause is constructed. If `reduction` then it is constructed lazily during the search. This is useful if the bottom clause

is too large to be constructed prior to search. This also sets the flag `lazy_bottom` to `true`. The user has to provide a refinement operator definition (using `refine/2`). If not, the `refine` parameter is set to `auto`. If `false` then no bottom clause is constructed. The user would normally provide a refinement operator definition in this case.

`set(dependent,+V)`
: *V* is a positive integer. Denotes the argument of the dependent variable in the examples (see Section 3.6 [Tree Learning], page 28 and Section 3.10 [Feature Construction], page 33).

`set(depth,+V)`
: *V* is a positive integer (default 10). Sets an upper bound on the proof depth to which theorem-proving proceeds.

`set(explore,+V)`
: *V* is one of: `true` or `false` (default `false`). If `true` then forces search to continue until the point that all remaining elements in the search space are definitely worse than the current best element (normally, search would stop when it is certain that all remaining elements are no better than the current best. This is a weaker criterion.) All internal pruning is turned off (see Section 3.2.3 [Pruning], page 22).

`set(evalfn,+V)`
: *V* is one of: `coverage`, `compression`, `posonly`, `pbayes`, `accuracy`, `laplace`, `auto_m`, `mestimate`, `entropy`, `gini`, `sd`, `wracc`, or `user` (default `coverage`). Sets the evaluation function for a search. See Section 3.2 [Other Searches], page 19.

`set(good,+V)`
: *V* is one of: `true` or `false` (default `false`). If `true` then stores a Prolog encoding of "good" clauses found in the search. A good clause is any clause with utility above that specified by the setting for `minscore`. If `goodfile` is set to some filename then this encoding is stored externally in that file.

`set(goodfile,+V)`
: *V* is a Prolog atom. Sets the filename for storing a Prolog encoding of good clauses found in searches conducted to date. Any existing file with this name will get appended.

`set(gsamplesize,+V)`
: *V* is a positive integer (default 100). The size of the randomly generated example set produced for learning from positive examples only. See Section 3.2 [Other Searches], page 19.

`set(i,+V)`
: *V* is a positive integer (default 2). Set upper bound on layers of new variables.

`set(interactive,+V)`
: *V* is one of: `true` or `false` (default `false`). If `true` then constructs theories interactively with `induce_rules` and `induce_tree`.

`set(language,+V)`
: *V* is an integer >= 1 or `inf` (default `inf`). Specifies the number of occurences of a predicate symbol in any clause.

`set(lazy_on_contradiction,+V)`
: *V* is one of: `true` or `false` (default `false`). Specifies if theorem-proving should proceed if a constraint is violated.

`set(lazy_on_cost,+V)`
: *V* is one of: `true` or `false` (default `false`). Specifies if user-defined cost-statements require clause coverages to be evaluated. This is normally not user-set, and decided internally.

`set(lazy_negs,+V)`
: *V* is one of: `true` or `false` (default `false`). If `true` then theorem-proving on negative examples stops once bounds set by `noise` or `minacc` are violated.

`set(lookahead,+V)`
: *V* is a positive integer. Sets a lookahead value for the automatic refinement operator (obtained by setting `refine` to `auto`).

`set(m,+V)`
: *V* is a floating point number. Sets a value for "m-estimate" calculations. See Section 3.2.2 [Search Function], page 21.

`set(max_abducibles,+V)`
: *V* is a positive integer (default `2`). Sets an upper bound on the maximum number of ground atoms within any abductive explanation for an observation. See Section 3.9 [Abductive Learning], page 32.

`set(max_features,+V)`
: *V* is a positive integer (default `inf`). Sets an upper bound on the maximum number of boolean features constructed by searching for good clauses. See Section 3.10 [Feature Construction], page 33

`set(minacc,+V)`
: *V* is an floating point number between 0 and 1 (default 0.0). Set a lower bound on the minimum accuracy of an acceptable clause. The accuracy of a clause has the same meaning as precision: that is, it is *p/(p+n)* where *p* is the number of positive examples covered by the clause (the true positives) and *n* is the number of negative examples covered by the clause (the false positives).

`set(mingain,+V)`
: *V* is an floating point number (default `0.05`). Specifies the minimum expected gain from splitting a leaf when constructing trees.

`set(minpos,+V)`
: *V* is a positive integer (default 1). Set a lower bound on the number of positive examples to be covered by an acceptable clause. If the best clause covers positive examples below this number, then it is not added to the current theory. This can be used to prevent Aleph from adding ground unit clauses to the theory (by setting the value to 2). Beware: you can get counter-intuitive results in conjunction with the `minscore` setting.

`set(minposfrac,+`*V*`)`
: *V* is a is a floating point number in the interval [0.0,1.0] (default 0.0). Set a lower bound on the positive examples covered by an acceptable clause as a fraction of the positive examples covered by the head of that clause. If the best clause has a ratio below this number, then it is not added to the current theory. Beware: you can get counter-intuitive results in conjunction with the `minpos` setting.

`set(minscore,+`*V*`)`
: *V* is an floating point number (default `-inf`). Set a lower bound on the utility of of an acceptable clause. When constructing clauses, If the best clause has utility below this number, then it is not added to the current theory. Beware: you can get counter-intuitive results in conjunction with the `minpos` setting.

`set(moves,+`*V*`)`
: *V* is an integer >= 0. Set an upper bound on the number of moves allowed when performing a randomised local search. This only makes sense if `search` is set to `rls` and `rls_type` is set to an appropriate value.

`set(newvars,+`*V*`)`
: *V* is a positive integer or `inf` (default `inf`). Set upper bound on the number of existential variables that can be introduced in the body of a clause.

`set(nodes,+`*V*`)`
: *V* is a positive integer (default 5000). Set upper bound on the nodes to be explored when searching for an acceptable clause.

`set(noise,+`*V*`)`
: *V* is an integer >= 0 (default 0). Set an upper bound on the number of negative examples allowed to be covered by an acceptable clause.

`set(openlist,+`*V*`)`
: *V* is an integer >= 0 or `inf` (default `inf`). Set an upper bound on the beam-width to be used in a greedy search.

`set(optimise_clauses,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). If `true` performs query optimisations described by V.S. Costa, A. Srinivasan, and R.C. Camacho in *A note on two simple transformations for improving the efficiency of an ILP system.*

`set(portray_examples,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). If `true` executes goal `aleph_portray(Term)` where `Term` is one of `train_pos`, `train_neg`, `test_pos`, or `test_neg` when executing the command `show(Term)`.

`set(portray_hypothesis,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). If `true` executes goal `aleph_portray(hypothesis)`. This is to be written by the user.

`set(portray_literals,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). If `true` executes goal `aleph_portray(Literal)` where `Literal` is some literal. This is to be written by the user.

`set(portray_search,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). If `true` executes goal `aleph_portray(search)`. This is to be written by the user.

`set(print,+`*V*`)`
: *V* is a positive integer (default 4). Sets an upper bound on the maximum number of literals displayed on any one line of the trace.

`set(proof_strategy,+`*V*`)`
: *V* is one of: `restricted_sld` or `sld` (default `restricted_sld`). If `restricted_sld`, then examples covered are determined by forcing current hypothesised clause to be the first parent clause in a SLD resolution proof. If `sld` then this restriction is not enforced. The former strategy is efficient, but not refutation complete. It is sufficient if all that is needed is to determine how many examples are covered by the current clause, which is what is needed when Aleph is used to construct a set of non-recursive clauses greedily (for example using the `induce/0` command: see Section 2.6 [Construct Theory], page 6).

`set(prooftime,+`*V*`)`
: *V* is a positive integer or `inf` (default `inf`). Sets an upper bound on the time (in seconds) for testing whether an example is covered. Overrides any value set for `searchtime`.

`set(prune_tree,+`*V*`)`
: *V* is is one of: `true` or `false` (default `false`). Determines whether rules constructed by the tree learner are subject to pessimistic pruning (see Section 3.6 [Tree Learning], page 28).

`set(record,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). If `true` then trace of Aleph execution is written to a file. The filename is given by `recordfile`.

`set(recordfile,+`*V*`)`
: *V* is a Prolog atom. Sets the filename to write a trace of execution. Only makes sense if `record` is set to `true`.

`set(refine,+`*V*`)`
: *V* is one of: `user`, `auto`, or `false` (default `false`). Specifies the nature of the customised refinement operator. In all cases, the resulting clauses are required to subsume the bottom clause, if one exists. If `false` then no customisation is assumed and standard operation results. If `user` then the user specifies a domain-specific refinement operator with `refine/2` statements. If `auto` then an automatic enumeration of all clauses in the mode language (see Section 2.2.1 [Modes], page 3) is performed. The result is a breadth-first branch-and-bound search starting from the empty clause. This is useful if a bottom clause is either not constructed or is constructed lazily. No attempt is made to ensure any kind of optimality and the same clauses may result from several different refinement paths. Some rudimentary checking can be achieved by setting `caching` to `true`. The user has to ensure the following for `refine` is set to `auto`: (1) the setting to `auto` is done after the modes and determinations commands, as these are

used to generate internally a set of clauses that allow enumeration of clauses in the language; (2) all arguments that are annotated as **#T** in the modes contain generative definitions for type **T**. These are called be the clauses generated internally to obtain the appropriate constants; and (3) the head mode is clearly specified using the `modeh` construct.

`set(rls_type,+`*V*`)`
: *V* is one of: `gsat`, `wsat`, `rrr`, or `anneal`. Sets the randomised search method to be one of GSAT, WSAT, RRR or simulated annealing. Requires `search` to be set to `rls`, and integer values for `tries` and `moves`. See Section 3.3 [Randomised Search], page 24.

`set(rulefile,+`*V*`)`
: *V* is a Prolog atom. Sets the filename for storing clauses found in theory (used by `write_rules/0`).

`set(samplesize,+`*V*`)`
: *V* is an integer >= 0 (default 0). Sets number of examples selected randomly by the `induce` or `induce_cover` commands. The best clause from the sample is added to the theory. A value of 0 turns off random sampling, and the next uncovered example in order of appearance in the file of training examples is selected.

`set(scs_percentile,+`*V*`)`
: *V* is an number in the range (0,100] (usually an integer). This denotes that any clause in the top *V*-percentile of clauses are considered "good" when performing stochastic clause selection. Only meaningful if `search` is set to `scs`.

`set(scs_prob,+`*V*`)`
: *V* is an number in the range [0,1.0). This denotes the minimum probability of obtaining a "good" clause when performing stochastic clause selection. Only meaningful if `search` is set to `scs`.

`set(scs_sample,+`*V*`)`
: *V* is a positive integer that determines the number of clauses randomly selected from the hypothesis space in a clause-level search. Only meaningful if `search` is set to `scs`. his overrules any samplesizes calculated from settings for `scs_percentile` and `scs_prob`.

`set(search,+`*V*`)`
: *V* is one of: `bf`, `df`, `heuristic`, `ibs`, `ils`, `rls`, `scs id`, `ic`, or `ar` (default `bf`). Sets the search strategy. See Section 3.2 [Other Searches], page 19.

`set(searchtime,+`*V*`)`
: *V* is an integer >= 0 or `inf` (default `inf`). Sets an upper bound on the time (in seconds) for a search.

`set(skolemvars,+`*V*`)`
: *V* is an integer (default 10000). Sets the counter for variables in non-ground positive examples. Each variable will be replaced by a skolem variable that has a unique number which is no smaller than *V*. This number has to be larger than the number of variables that would otherwise appear in a bottom clause.

`set(splitvars,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). If set to `true` before constructing a bottom clause, then variable co-references in the bottom clause are split apart by new variables. The new variables can occur at input or output positions of the head literal, and only at output positions in body literals. Equality literals between new and old variables are inserted into the bottom clause to maintain equivalence. It may also result in variable renamed versions of other literals being inserted into the bottom clause. All of this increases the search space considerably and can make the search explore redundant clauses. The current version also elects to perform variable splitting whilst constructing the bottom clause (in contrast to doing it dynamically whilst searching). This was to avoid unnecessary checks that could slow down the search when variable splitting was not required. This means the bottom clause can be extremely large, and the whole process is probably not very practical for large numbers of co-references. The procedure has not been rigourously tested to quantify this.

`set(stage,+`*V*`)`
: *V* is one of: `saturation`, `reduction` or `command` (default `command`). Sets the stage of current execution. This is normally not user-set, and decided internally.

`set(store_bottom,+`*V*`)`
: *V* is one of: `true` or `false` (default `false`). Stores bottom clause constructed for an example for future re-use.

`set(temperature,+`*V*`)`
: *V* is a non-zero floating point number. Sets the temperature for randomised search using annealing. Requires `search` to be set to `rls` and `rls_type` to be set to `anneal`.

`set(test_pos,+`*V*`)`
: *V* is a Prolog atom or a list of Prolog atoms. Sets the filename or list of filenames containing the positive examples for testing. No filename extensions are assumed and complete filenames have to be provided.

`set(test_neg,+`*V*`)`
: *V* is a Prolog atom or a list of Prolog atoms. Sets the filename or list of filenames containing the negative examples for testing. No filename extensions are assumed and complete filenames have to be provided.

`set(threads,+`*V*`)`
: *V* is an integer >= 1 (default 1). This is experimental and should not be changed from the default value until further notice.

`set(train_pos,-`*V*`)`
: *V* is a Prolog atom or a list of Prolog atoms. Sets the filename or list of filenames containing the positive examples. If set, no filename extensions are assumed and complete filenames have to be provided. If not set, it is internally assigned a value after the `read_all` command.

`set(train_neg,-`*V*`)`
: *V* is a Prolog atom or a list of Prolog atoms. Sets the filename or list of filenames containing the negative examples. If set, no filename extensions are

assumed and complete filenames have to be provided. If not set, it is internally assigned a value after the `read_all` command.

`set(tree_type,+V)`
: *V* is one of `classification`, `class_probability`, `regression`, or `model` (see (see Section 3.6 [Tree Learning], page 28).

`set(tries,+V)`
: *V* is a positive integer. Sets the maximum number of restarts allowed for randomised search methods. This only makes sense if `search` is set to `rls` and `rls_type` is set to an appropriate value.

`set(typeoverlap,+V)`
: *V* is a floating point number in the interval (0.0,1.0]. Used by `induce_modes/0` to determine if a pair of different types should be given the same name. See Section 3.8 [Mode Learning], page 31 for more details.

`set(uniform_sample,+V)`
: *V* is one of: `true` or `false` (default `false`). Used when drawing clauses randomly from the clause-space. If set set to `true` then clauses are drawn by uniform random selection from the space of legal clauses. Since there are usually many more longer clauses than shorter ones, this will mean that clauses drawn randomly are more likely to be long ones. If set to `false` then assumes a uniform distribution over clause lengths (up to the maximum length allowed by `clauselength`). This is not necessarily uniform over legal clauses. If random clause selection is done without a bottom clause for guidance then this parameter is set to `false`.

`set(updateback,+V)`
: *V* is one of: `true` or `false` (default `true`). If `false` then clauses found by the `induce` family are not incorporated into the background. This is experimental.

`set(verbosity,+V)`
: *V* is an integer >= 0 (default 1). Sets the level of verbosity. Also sets the parameter `verbose` to the same value. A value of 0 shows very little.

`set(version,-V)`
: *V* is the current version of Aleph. This is set internally.

`set(walk,+V)`
: *V* is a value between `0` and `1`. It represents the random walk probability for the Walksat algorithm.

`set(+P,+V)`
: Sets any user-defined parameter *P* to value *V*. This is particularly useful when attaching notes with particular experiments, as all settings can be written to a file (see `record`). For example, `set(experiment,'Run 1 with background B0')`.

3.2 Altering the search

Aleph allows the basic procedure for theory construction to be altered in a number of ways. Besides the `induce` command, there are several other commands that can be used to construct a theory. The `induce` family of commands are:

1. `induce/0`. This has already been described in detail previously (see Section 2.6 [Construct Theory], page 6);
2. `induce_cover/0`. This command is very similar to `induce`. The only difference is that positive examples covered by a clause are not removed prior to seeding on a new (uncovered) example. After a search with `induce_cover` Aleph only removes the the examples covered by the best clause are removed from a pool of seed examples only. After this, a new example or set of examples is chosen from the seeds left, and the process repeats. The theories returned by `induce` and `induce_cover` are dependent on the order in which positive examples are presented;
3. `induce_max/0`. The theory returned by this command is unaffected by the ordering of positive examples. This is because it saturates and reduces every example. The search is made more efficient by remembering the coverage of the best clause obtained so far for each example being generalised. Both `induce_cover` and `induce_max` are slower than `induce`, and usually produce clauses with a great deal of overlap in coverage. A separate program will have to be used to find some subset of these clauses that minimises this overlap (see *T-Reduce* in Chapter 4 [Other Programs], page 39).
4. `induce_incremental/0`. This command constructs a theory in an incremental mode: the user is allowed to update the examples and background knowledge. This mode of learning is described further in Section 3.4 [Incremental Learning], page 26.
5. `induce_clauses/0`. This command is simply `induce/0` or `induce_incremental/0` depending on whether the flag `interactive` is `false` or `true`.
6. `induce_theory/0`. This command abandons the clause-by-clause approach to theory construction. Instead, search is done at the theory-level. This is untested and the current implementation should not be considered definitive. See Section 3.5 [Theory Learning], page 27 for more details.
7. `induce_tree/0`. This command abandons the clause-by-clause approach to theory construction. Instead, search is done by constructing a tree using the standard recursive-partitioning approach. See Section 3.6 [Tree Learning], page 28 for more details.
8. `induce_constraints/0`. This command abandons the search for predictive clauses. Instead, search results in all constraints that hold within the background knowledge provided. See Section 3.7 [Constraint Learning], page 30 for more details.
9. `induce_modes/0`. This command searches for a mode and type assignment that is consistent with the background knowledge provided. See Section 3.8 [Mode Learning], page 31 for more details.
10. `induce_features/0`. This command searches for boolean features given the examples and the background knowledge. See Section 3.10 [Feature Construction], page 33 for more details.

The search for individual clauses (when performed) is principally affected by two parameters. One sets the search strategy (`search`) and the other sets the evaluation function (`evalfn`).

3.2.1 Search strategies

A search strategy is set using `set(search,Strategy)`. The following search strategies apply to the clause-by-clause searches conducted by Aleph:

`ar`
: Implements a simplified form of the type of association rule search conducted by the WARMR system (see L. Dehaspe, 1998, PhD Thesis, Katholieke Universitaet Leuven). Here, Aleph simply finds all rules that cover at least a pre-specified fraction of the positive examples. This fraction is specified by the parameter `pos_fraction`.

`bf`
: Enumerates shorter clauses before longer ones. At a given clauselength, clauses are re-ordered based on their evaluation. This is the default search strategy;

`df`
: Enumerates longer clauses before shorter ones. At a given clauselength, clauses are re-ordered based on their evaluation.

`heuristic`
: Enumerates clauses in a best-first manner.

`ibs`
: Performs an iterative beam search as described by Quinlan and Cameron-Jones, IJCAI-95. Limit set by value for `nodes` applies to any 1 iteration.

`ic`
: Performs search for integrity constraints. Used by `induce_constraints` (see Section 3.7 [Constraint Learning], page 30)

`id`
: Performs an iterative deepening search up to the maximum clause length specified.

`ils`
: An iterative `bf` search strategy that, starting from 1, progressively increases the upper-bound on the number of occurrences of a predicate symbol in any clause. Limit set by value for `nodes` applies to any 1 iteration. This language-based search was developed by Rui Camacho and is described in his PhD thesis.

`rls`
: Use of the GSAT, WSAT, RRR and simulated annealing algorithms for search in ILP. The choice of these is specified by the parameter `rls_type` (see Section 3.1 [Other Settings], page 10). GSAT, RRR, and annealing all employ random multiple restarts, each of which serves as the starting point for local moves in the search space. A limit on the number of restarts is specified by the parameter `tries` and that on the number of moves by `moves`. Annealing is currently restricted to a using a fixed temperature, making it equivalent to an algorithm due to Metropolis. The temperature is specified by setting the parameter `temperature`. The implementation of WSAT requires a "random-walk probability", which is specified by the parameter `walk`. A walk probability of 0 is equivalent to GSAT. More details on randomised search can be found in Section 3.3 [Randomised Search], page 24.

`scs`
: A special case of GSAT that results from repeated random selection of clauses from the hypothesis space. The number of clauses is either set by `scs_sample` or is calculated from the settings for `scs_prob` and `scs_percentile`. These represent: the minimum probability of selecting a "good" clause; and the meaning of a "good" clause, namely, that it is in the top K-percentile of clauses. This invokes GSAT search with `tries` set to the sample size and `moves` set to 0.

Clause selection can either be blind or informed by some preliminary Monte-Carlo style estimation. This is controlled by `scs_type`. More details can be found in Section 3.3 [Randomised Search], page 24.

3.2.2 Evaluation functions

An evaluation function is set using `set(evalfn,Evalfn)`. The following clause evaluation functions are recognised by Aleph:

`accuracy`
: Clause utility is `P/(P+N)`, where `P`, `N` are the number of positive and negative examples covered by the clause.

`auto_m`
: Clause utility is the m estimate (see `mestimate` below) with the value of `m` automatically set to be the maximum likelihood estimate of the best value of `m`.

`compression`
: Clause utility is `P - N - L + 1`, where `P`, `N` are the number of positive and negative examples covered by the clause, and `L` the number of literals in the clause.

`coverage`
: Clause utility is `P - N`, where `P`, `N` are the number of positive and negative examples covered by the clause.

`entropy`
: Clause utility is `p log p + (1-p) log (1-p)` where `p = P/(P + N)` and `P`, `N` are the number of positive and negative examples covered by the clause.

`gini`
: Clause utility is `2p(1-p)` where `p = P/(P + N)` and `P`, `N` are the number of positive and negative examples covered by the clause.

`laplace`
: Clause utility is `(P+1)/(P+N+2)` where `P`, `N` are the positive and negative examples covered by the clause.

`mestimate`
: Clause utility is its m estimate as described in S. Dzeroski and I. Bratko (1992), *Handling Noise in Inductive Logic Programming*, Proc. Second Intnl. Workshop on Inductive Logic Programming, ICOT-TM-1182, Inst. for New Gen Comput Technology, Japan. The value of `m` is set by `set(m,M)`.

`pbayes`
: Clause utility is the pseudo-Bayes conditional probability of a clause described in J. Cussens (1993), *Bayes and Pseudo-Bayes Estimates of Conditional Probability and their Reliability*, ECML-93, Springer-Verlag, Berlin.

`posonly`
: Clause utility is calculated using the Bayesian score described in S. H. Muggleton, (1996), *Learning from positive data*, Proc. Sixth Intnl. Workshop on Inductive Logic Programming, LNAI 1314, 358-376, Springer-Verlag, Berlin. Note that all type definitions are required to be generative for this evaluation function and a `modeh` declaration is necessary.

`sd`
: Clause utility is related to the standard deviation of values predicted. This is only used when constructing regression trees and is not available for use during clause-based search.

`user`
: Clause utility is `-C`, where `C` is the value returned by a user-defined cost function. See Section 3.2.4 [Cost], page 22.

`wracc` Clause utility is calculated using the weighted relative accuracy function described by N. Lavrac, P. Flach and B. Zupan, (1999), *Rule Evaluation Measures: a Unifying View*, Proc. Ninth Intnl. Workshop on Inductive Logic Programming, LNAI 1634, 174-185, Springer-Verlag, Berlin.

3.2.3 Built-in and user-defined pruning

Two sorts of pruning can be distinguished within Aleph when performing a clause-level search. Internal pruning refers to built-in pruning that performs admissible removal of clauses from a search. This is currently available for the following evaluation functions: auto_m, compression, coverage, laplace, mestimate, posonly, and wracc. User-defined prune statements can be written to specify the conditions under which a user knows for certain that a clause (or its refinements) could not possibly be an acceptable hypothesis. Such clauses are pruned from the search. The "prune" definition is written in the background knowledge file (that has extension `.b`). The definition is distinguished by the fact that they are all rules of the form:

```
prune((ClauseHead:-ClauseBody)) :-
        Body.
```

The following example is from a pharmaceutical application that states that every extension of a clause representing a "pharmacophore" with six "pieces" is unacceptable, and that the search should be pruned at such a clause.

```
prune((Head:-Body)) :-
        violates_constraints(Body).

violates_constraints(Body) :-
        has_pieces(Body,Pieces),
        violates_constraints(Body,Pieces).

violates_constraints(Body,[_,_,_,_,_,_]).

has_pieces(...) :-
```

The use of such pruning can greatly improve Aleph's efficiency. They can be seen as a special case of providing distributional information about the hypothesis space.

3.2.4 User-defined cost specification

The use of a user-specified cost function is a fundamental construct in statistical decision theory, and provides a general method of scoring descriptions. Aleph allows the specification of the cost of a clause. The cost statements are written in the background knowledge file (that has extension `.b`), and are distinguished by the fact that they are all rules of the form:

```
cost(Clause,ClauseLabel,Cost):-
        Body.
```

where `ClauseLabel` is the list `[P,N,L]` where `P` is the number of positive examples covered by the clause, `N` is the number of negative examples covered by the clause `L` is the number of literals in the clause.

It is usually not possible to devise automatically admissible pruning strategies for an arbitrary cost function. Thus, when using a user-defined cost measure, Aleph places the burden of specifying a pruning strategy on the user.

3.2.5 User-defined constraints

Aleph accepts integrity constraints that should not be violated by a hypothesis. These are written in the background knowledge file (that has extension `.b`) and are similar to the integrity constraints in the ILP programs Clint and Claudien. The constraints are distinguished by the fact that they are all rules of the form:

```
false:-
         Body.
```

where `Body` is a set of literals that specify the condition(s) that should not be violated by a clause found by Aleph. It is usual to use the `hypothesis/3` (see Section 3.11 [Other Commands], page 34) command to obtain the clause currently being considered by Aleph.

The following example is from a pharmaceutical application that states that hypotheses are unacceptable if they have fewer than three "pieces" or which do not specify the distances between all pairs of pieces.

```
false:-
        hypothesis(Head,Body,_),
        has_pieces(Body,Pieces),
        length(Pieces,N),
        N =< 2.
false:-
        hypothesis(_,Body,_),
        has_pieces(Body,Pieces),
        incomplete_distances(Body,Pieces).
```

The use of constraints is another way for Aleph to obtain interesting hypothesis without negative examples. Ordinarily, this will result in a single clause that classifies every example as positive. Such clauses can be precluded by constraints. Note also that an integrity constraint does not state that a refinement of a clause that violates one or more constraints will also be unacceptable. When constructing clauses in an incremental mode, Aleph can be instructed to add a special type of constraint to prevent the construction of overly general clauses (see Section 3.4 [Incremental Learning], page 26).

3.2.6 User-defined refinement

Aleph allows a method of specifying the refinement operator to be used in a clause-level search. This is done using a Prolog definition for the predicate `refine/2`. The definition specifies the transitions in the refinement graph traversed in a search. The "refine" definition is written in the background knowledge file (that has extension ".b"). The definition is distinguished by the fact that they are all rules of the form:

```
refine(Clause1,Clause2):-
          Body.
```

This specifies that Clause1 is refined to Clause2. The definition can be nondeterministic, and the set of refinements for any one clause are obtained by repeated backtracking. For any refinement Aleph ensures that Clause2 implies the current most specific clause. Clause2 can contain cuts ("!") in its body.

The following example is from a pharmaceutical application that states that searches for a "pharmacophore" that consists of 4 "pieces" (each piece is some functional group), and associated distances in 3-D space. Auxilliary definitions for predicates like member/2 and dist/5 are not shown. representing a "pharmacophore" with six "pieces" is unacceptable, and that the search should be pruned at such a clause.

```
refine(false,active(A)).

refine(active(A),Clause):-
        member(Pred1,[hacc(A,B),hdonor(A,B),zincsite(A,B)]),
        member(Pred2,[hacc(A,C),hdonor(A,C),zincsite(A,C)]),
        member(Pred3,[hacc(A,D),hdonor(A,D),zincsite(A,D)]),
        member(Pred4,[hacc(A,E),hdonor(A,E),zincsite(A,E)]),
        Clause = (active(A):-
                        Pred1,
                        Pred2,
                        dist(A,B,C,D1,E1),
                        Pred3,
                        dist(A,B,D,D2,E2),
                        dist(A,C,D,D3,E3),
                        Pred4,
                        dist(A,B,E,D4,E4),
                        dist(A,C,E,D5,E5),
                        dist(A,D,E,D6,E6)).
```

To invoke the use of such statements requires setting `refine` to `user`. For other settings of `refine` see entry for `refine` in Section 3.1 [Other Settings], page 10.

3.3 Randomised search methods

The simplest kind of randomised search is the following: sample N elements (clauses or theories) from the search space. Score these and return the best element. Ordinal optimisation is a technique that investigates the loss in optimality resulting from this form of search. See:

`http://hrl.harvard.edu/people/faculty/ho/DEDS/OO/OOTOC.html`

A study of the use of this in ILP can be found in: A. Srinivasan, *A study of two probabilistic methods for searching large spaces with ILP* (under review), available at:

`ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Papers/AS/dami99.ps.gz`

For a clause-level search, this is invoked by setting the parameter `search` to `scs` (to denote "stochastic clause selection"). The number N is either set by assigning a value to `scs_sample` or calculated automatically from settings for `scs_prob` and `scs_percentile`. If these values are denoted "P" and "K" respectively, then the sample size is calculated to be `log(1-P)/log(1-K/100)`, which denotes the number of clauses that have to be sampled before obtaining, with probability at least P, at least one clause in the top K-percentile of clauses Sampling is further controlled by by specifying the setting `scs_type` to be one of `blind` or `informed`. If “blind” then clauses are uniform random selections from the space of all legal clauses. If “informed” then they are drawn from a specific distribution over clauselengths. This can either be pre-specified (by setting `clauselength_distribution`) or obtained automatically by a Monte-Carlo like scheme that attempts to estimate, for each clause length, the probablity of obtaining a clause in the top K-percentile. In either case, the resulting distribution over clauselengths is used to first decide on the number of literals “l” in the clause. A legal clause with “l” literals is then constructed.

In fact, this simple randomised search is a degenerate form of a more general algorithm known as GSAT. Originally proposed within the context of determining satisfiability of propositional formulae, the basic algorithm is as follows:

```
currentbest:= 0 (comment: ‘‘0’’ is a conventional default answer)
for i = 1 to N do
   current:= randomly selected starting point
   if current is better than currenbest then
        currentbest:= current
   for j = 1 to M do begin
        next:= best local move from current
        if next is better than currenbest then
            currentbest:= next
        current:= next
   end
return currentbest
```

`N` and `M` represent the number of tries and moves allowed. It is apparent that when searching for clauses, a `M` value of 0 will result in the algorithm mimicking stochastic clause selection as described above. A variant of this algorithm called Walksat introduces a further random element at the point of selecting `next`. This time, a biased coin is flipped. If a “head” results then the choice is as per GSAT (that is, the best choice amongst the local neighbours), otherwise `next` is randomly assigned to one of any “potentially good” neighbours. Potentially good neighbours are those that *may* lead to a better score than the current best score. This is somewhat like simulated annealing, where the choice is the best element if that improves on the best score. Otherwise, the choice is made according to a function that decays exponentially with the difference in scores. This exponential decay is usually weighted by a “temperature” parameter.

The randomly selected start clause is usually constructed as follows: (1) an example is selected; (2) the bottom clause is constructed for the example; (3) a legal clause is randomly drawn from this bottom clause. The example may be selected by the user (using the `sat` command). If bottom clauses are not allowed (by setting `construct_bottom` to `false`) then legal clauses are constructed directly from the mode declarations. The clause selected is either the result of uniform random selection from all legal clauses, or the result of a specific distribution over clauselengths (specified by setting `clauselength_distribution`). The latter is the only method permitted when bottom clauses are not allowed. (In that case, if there is no value specified for `clauselength_distribution`, then a uniform distribution over all allowable lengths is used.)

RRR refers to the ‘randomised rapid restarts’ as described by F. Zelezny, A. Srinivasan, and D. Page in *Lattice Search Runtime Distributions May Be Heavy-Tailed* available at:

`ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Papers/AS/rrr.ps.gz`

In the current implementation, RRR stops as soon as a clause with an requisite minimum positive coverage (set using `minpos`) and acceptable utility is reached (set using `minscore`). The procedure in the paper above stops as soon as a minimum acceptable accuracy is reached. This same effect can be achieved by setting `evalfn` to `accuracy`.

It is intended that the randomised local search methods (GSAT, Walksat, RRR and annealing) can be used either for clause-level search or theory-level search. No equivalent of stochastic clause selection is provided for theory-level search: this has to be mimicked by using the randomised local search, with appropriate settings. At the clause level, local moves involve either adding or deleting a literal from the current clause. Normally, local moves in the clause-space would also involve operations on variables (introducing or removing variable co-references, associating or disassociating variables to constants). These have to accomplished within Aleph by the inclusion of an equality predicate with appropriate mode declarations. Local moves for a theory-level search are described in Section 3.5 [Theory Learning], page 27.

Randomised local search is invoked within Aleph by setting the parameter `search` to `rls`. In addition, the type of search is specified by setting `rls_type` to one of `gsat`, `wsat`, `rrr` or `anneal`. Walksat requires a specification of a biased coin. This is done by setting the parameter `walk` to a number between `0` and `1`. This represents an upper bound on the probability of obtaining a "tail" with the coin. The implementation of simulated annealing is very simple and uses a fixed temperature. This is done by setting the parameter `temperature` to some real value.

3.4 Incremental construction of theories

Most prominent ILP systems are "batch learners": all examples and background knowledge are in place *before* learning commences. The ILP system then constructs a hypothesis for the examples. A less popular, but nevertheless interesting alternative is that of "incremental learning", where examples, background and hypothesis are incrementally updated *during* the course of learning. Aleph allows such an incremental construction of clauses by typing:

```
induce_incremental.
```

This results in Aleph repeatedly performing the following steps:

1. **Ask user for an example.** The default is to use a new positive example from previous search. If the user responds with Ctrl-d (eof) then search stops. If the user responds with "ok." then default is used; otherwise the user has to provide a new example (terminated by a full-stop);
2. **Construct bottom clause for example.** Aleph thus expects the appropriate mode declarations. These can be added in Step 4;
3. **Search.** Aleph searches for the best clause;
4. **Ask user about best clause.** Aleph asks the user about the clause C returned by the search. At this point the user can respond with:
   - **ok.** Clause C is added to the hypothesis;
   - **prune.** Statement added to prevent C and any clauses subsumed by it from appearing as the result of future searches;
   - **overgeneral.** Constraint added to prevent C and clauses subsuming it from appearing as the result of future searches;

- **overgeneral because not E. E** is added as a negative example;
- **overspecific.** C is added as a positive example;
- **overspecific because E. E** is added as a positive example;
- **X. X** is any Aleph command. This can be something like `covers` or `mode(*,has_car(+train,-car))`;
- **Ctrl-d.** Returns to Step 1.

Note: the command `induce_clauses/0` with the flag `interactive` set to `true` simply performs the same function as `induce_incremental`.

The incremental mode does not preclude the use of prior sets of examples or background information. These are provided in the usual way (in files with `.b`, `.f` and `.n` suffixes). An example of using the incremental learner to construct a program for list membership can be found in the `incremental` sub-directory in:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

3.5 Theory-level search

An adequate explanation for a set of examples typically requires several clauses. Most ILP systems attempt to construct such explanations one clause at a time. The procedure is usually an iterative greedy set-covering algorithm that finds the best single clause (one that explains or "covers" most unexplained examples) on each iteration. While this has been shown to work satisfactorily for most problems, it is nevertheless interesting to consider implementations that attempt to search directly at the "theory-level". In other words, elements of the search space are sets of clauses, each of which can be considered a hypothesis for all the examples. The implementation in Aleph of this idea is currently at a very rudimentary level, and preliminary experiments have not demonstrated great benefits. Nevertheless, the approach, with development, could be promising. The implementation within Aleph is invoked by the command:

```
induce_theory.
```

This conducts a search that moves from one set of clauses to another. Given a clause set S local moves are the result of the following:

1. **Add clause.** A clause is added to S. This is usually a randomly selected legal clause constructed in the manner described in Section 3.3 [Randomised Search], page 24;
2. **Delete clause.** A clause is deleted from S;
3. **Add literal.** A literal is added to a clause in S; and
4. **Delete literal.** A literal is deleted from a clause in S.

As noted in Section 3.3 [Randomised Search], page 24, the use of an equality predicate with appropriate mode declarations may be needed to achieve variable co-references, etc.

Currently, `induce_cover` starts with an initial set of at most C clauses, where this number is specified by setting the `clauses` parameter. Each of these are randomly selected legal clauses. `induce_cover` then performs theory-level search either using as search strategy a randomised local search method (obtained by setting the `search` parameter to `rls`: see Section 3.3 [Randomised Search], page 24), or a markov chain monte carlo technique (obtained by setting `search` to `mcmc`). The latter is untested. The only evaluation function allowed is `accuracy`. For theories, this is the number `(TP+TN)/(TP+TN+FP+FN)` where `TP,TN` are are the numbers of positive and negative examples correctly classified respectively; `FP` is the numbers of negative examples incorrectly classified as positive; and `FN` is the number of positive examples incorrectly classified as positive.

3.6 Tree-based theories

The algorithm embodied in `induce` can be seen as the first-order equivalent of a propositional rule-learning algorithms like Clark and Niblett's CN2. There is now a substantial body of empirical work (done by researchers in Leuven and Freiburg) demonstrating the utility of first-order equivalents of propositional tree-learning procedures. Tree-based learning can be seen as a special case of theory learning and the implementation in Aleph uses the standard recursive-partitioning approach to construct classification, regression, class probability, or model trees. Tree-based theory construction is invoked by the command:

```
induce_tree.
```

The type of tree constructed is determined by setting `tree_type` to one of: `classification`, `regression`, `class_probability`, or `model`. The basic procedure attempts to construct a tree to predict the output argument in the examples. Note that the mode declarations must specify only a single argument as output. Paths from root to leaf constitute clauses. Tree-construction is viewed as a refinement operation: any leaf can currently be refined (converted into a non-leaf) by extending the corresponding clause (resulting in two new leaves). The extension is done using Aleph's automatic refinement operator that extends clauses by a single literal within the mode language . That is, Aleph sets `refine` to `auto`. Note that using the automatic refinement operator means that the user has to ensure that all arguments that are annotated as **#T** in the modes contain generative definitions for type **T**. The `lookahead` option allows additions of several literals at once. The impurity function is specified by the setting the `evalfn` parameter. Currently for `classification` and `class_probability` trees `evalfn` must be one of `entropy` or `gini`. For `regression` trees the evaluation function is automatically set to `sd` (standard deviation). For `model` trees, `evalfn` must be one of `mse` (mean square error) or `accuracy`. In all cases, the result is always presented a set of rules. Rules for `class_probability` and `regression` trees make their predictions probabilistically using the `random/2` predicate provided within Aleph.

In addition, settings for the following parameters are relevant: `classes`, the list of classes occuring in examples provided (for `classification` or `class_probability` trees only); `dependent`, for the argument constituting the dependent variable in the examples; `prune_tree`, for pruning rules from a tree; `confidence`, for error-based pruning of rules as described by J R Quinlan in the C4.5 book; `lookahead`, specifying the lookahead for the refinement operator to mitigate the horizon effect from zero-gain literals; `mingain`, specifying the minimum gain required for refinement to proceed; and `minpos` specifying the minimum number of examples required in a leaf for refinement to proceed.

Forward pruning is achieved by the parameters (`mingain`) and `minpos`. The former should be set to some value greater than 0 and the latter to some value greater than 1. Backward pruning uses error pruning of the final clauses in the tree by correcting error estimates obtained from the training data. Automatic error-based pruning is achieved by setting the parameter `prune_tree` to `auto`. For `classification` trees the resulting procedure is identical to the one for rule pruning described by Quinlan in C4.5: Programs for Machine Learning, Morgan Kauffmann. For `regression` trees, error-based pruning results in corrections to the sample standard deviation. These corrections assume normality of observed values in a leaf: the method has been studied emprically by L. Torgo in "A Comparative Study of Reliable Error Estimators for Pruning Regression Trees". Following work by F Provost and P Domingos, pruning is not employed for class probability prediction. At this stage, there is no pruning also for model trees.

The prediction at each 'leaf' differs for each tree type. For `classification` trees, prediction is the majority class as estimated from the examples in the leaf; for `regression` trees prediction is a value drawn randomly from a normal distribution with mean and standard deviation estimated from the examples in the leaf; for `class_probability` trees prediction is a value drawn randomly from the (Laplace corrected) discrete distribution of classes in the leaf; and for `model` trees prediction is achieved by a user-defined background predicate (see following).

Model trees in Aleph are constructed by examining, at each leaf, one or more model construction predicates. These predicates are defined as part of background knowledge, and can specify different kinds of models For example, the predicates may be for linear regression, polynomial regression etc. for predicting a continuous variable; a decision tree, logistic regression etc. for predicting a nominal variable. For each kind of model, the user has to provide a definition for a predicate that is able to: (a) construct the model; and (b) predict using the model constructed. The process is the same as that for lazy evaluation. Each such predicate is specified using the `model/1` command. If several different predicates are specified, then, at each leaf, each predicate is called to construct a model and the predicate that constructs the best model (evaluated using the current setting for `evalfn`) is returned. This can be computationally intensive, but can lead to the construction of fairly complex theories, in which different leaves can contain different kinds of models (for example, linear regression models in one leaf and quadratic regression models in another).

Tree-learning can be performed interactively, with the user specifying the split to be selected. This is done by setting `interactive` to `true` before executing the `induce_tree` command.

An example of using the tree learner can be found in the `tree` sub-directory in:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

3.7 Constraint learning

The basic Aleph algorithm constructs definite clauses normally intended to be components of a predictive model for data. Early ILP work (for example, in the Claudien system) demonstrated the value of discovering all non-Horn constraints that hold in a database. A similar functionality can be obtained within Aleph using the command:

```
    induce_constraints.
```

The implementation of these ideas in Aleph uses a naive generate-and-test strategy to enumerate all constraints within the background knowledge (for the mode language provided). All constraints are of the form:

```
false:- ...
```

and are stored in the user-specified `goodfile` (the specification of this file is mandatory for `induce_constraints` to work). With appropriate mode settings for `false` and `not` it is possible to identify non-Horn constraints in the same way as Claudien. For example given the background knowledge:

```
male('Fred').
female('Wilma').

human('Fred').
human('Wilma').
```

and the mode declarations:

```
:- modeh(1,false).

:- modeb(*,human(-person)).
:- modeb(1,male(+person)).
:- modeb(1,female(+person)).
:- modeb(1,not(male(+person))).
:- modeb(1,not(female(+person))).
```

Aleph identifies the following constraints:

```
false :-
    human(A), male(A), female(A).
false :-
    human(A), female(A), male(A).
false :-
    human(A), not male(A), not female(A).
false :-
    human(A), not female(A), not male(A).
```

After removing redundant constraints (which Aleph does not do), these are equivalent to the following:

```
false :- human(A), male(A), female(A).

male(A) ; female(A) :- human(A).
```

The validity of these constraints can only be guaranteed if the background knowledge is assumed to be complete and correct. To account for incorrect statements in the background knowledge it may sometimes be relevant to alter the `noise` setting when obtaining constraints which now specifies the number of falsifying substitutions tolerated. The `minacc` parameter is ignored.

An example of using the constraints learner can be found in the `constraints` subdirectory in:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

3.8 Mode learning

The basic Aleph algorithm assumes modes will be declared by the user which, in the past, this has been the source of some difficulty. There has been some work (by E. McCreath and A. Sharma, Proc of the 8th Australian Joint Conf on AI pages 75-82, 1995) on automatic extraction of mode and type information from the background knowledge provided. The implementation of these ideas in Aleph follows these ideas fairly closely and can be invoked by the command:

```
    induce_modes.
```

Given a set of determinations, the procedure works in two parts: (i) finding equivalence classes of types; and (ii) finding an input/output assignment.

Unlike the McCreath and Sharma approach, types in the same equivalence class are given the same name only if they "overlap" significantly (the overlap of type1 with type2 is the proportion of elements of type1 that are also elements of type2). Significantly here means an overlap at least some threshold T (set using `typeoverlap`, with default 0.95). Values of `typeoverlap` closer to 1.0 are more conservative, in that they require very strong overlap before the elements are called the same type. Since this may not be perfect, modes are also produced for equality statements that re-introduce co-referencing amongst differently named types in the same equivalence class. The user has to however explicitly include a determination declaration for the equality predicate.

The i/o assignment is not straightforward, as we may be dealing with non-functional definitions. The assignment sought here is one that maximises the number of input args as this gives the largest bottom clause. This assignment is is sought by means of a search procedure over mode sequences. Suppose we have a mode sequence `M = <m1,m2,..m\i-1\>` that uses the types T. An argument of type t in mode `m\i\` is an input iff t overlaps significantly (used in the same sense as earlier) with some type in T. Otherwise the argument is an output. The utility of each mode sequence M is f(M) = g(M) + h(M) where g(M) is the number of input args in M; and h(M) is a (lower) estimate of the number of input args in any mode sequence of which M is a prefix. The search strategy adopted is a simple hill-climbing one. Note that the procedure as implemented assumes background predicates will be generative (which holds when the background knowledge is ground).

An example of using the mode learner can be found in the `modes` sub-directory in:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

3.9 Abductive learning

The basic Aleph algorithm assumes that the examples provided are observations of the target predicate to be learned. There is, in fact, nothing within the ILP framework that requires this to be the case. For example, suppose the following was already provided in the background knowledge:

```
                    grandfather(X,Y):-
                        father(X,Z),
                        parent(Z,Y).

                    parent(X,Y):-
```

```
         father(X,Y).

    father('Fred','Jane').

    mother('Jane','Robert').
    mother('Jane','Peter').
```

then the examples:

```
    grandfather('Fred','Robert').
    grandfather('Fred','Peter').
```

are clearly not entailed by the background knowledge. Aleph would then simply try to learn another clause for `grandfather/2`, perhaps resulting in something like:

```
    grandfather(X,Y):-
         father(X,Z),
         mother(Z,Y).
```

In fact, the job would have just as easily been done, and the result more useful, if Aleph could learn the following:

```
    parent(X,Y):-
         mother(X,Y).
```

This requires Aleph to be able to do two things. First, given observations of `grandfather/2` that are not entailed by the background knowledge, generate instances of `parent/2` that will allow the observations to be entailed. Second, use the instances of `parent/2` that were generated to obtain the clause for `parent/2` above. The first of these steps requires a form of abduction. The second requires generalisation in the form of learning. It is the combination of these two steps that is called "Abductive Learning" here.

The basic procedure used by Aleph is a simplified variant of S. Moyle's Alecto program. Alecto is described in some detail in S. Moyle, "Using Theory Completion to Learn a Navigation Control Program", Proceedings of the Twelfth International Conference on ILP (ILP2002), S. Matwin and C.A. Sammut (Eds), LNAI 2583, pp 182-197, 2003. Alecto does the following: for each positive example, an "abductive explanation" is obtained. This explanation is set of ground atoms. The union of abductive explanations from all positive examples is formed (this is also a set of ground atoms). These are then generalised to give the final theory. The ground atoms in an abductive explanation are obtained using Yamamoto's SOLD resolution or SOLDR (Skip Ordered Linear resolution for Definite clauses).

Currently, abductive learning is only incorporated within the `induce` command. If `abduce` is set to `true` then Aleph first tries to obtain the best clause for the observed predicate (for example, the best clause for `grandfather/2`). Abductive explanations are then generated for all predicates marked as being abducible (see `abducible/1`) and generalisations constructed using these. The best generalisation overall is then selected and greedy clause identification by `induce` repeats with the observations left. Care has to be taken to ensure that abductive explanations are indeed ground (this can be achieved by using appropriate type predicates within the definitions of the abducible predicates) and limited to some maximum number (this latter requirement is for reasons of efficiency: see setting for `max_abducibles`).

It should be evident that abductive learning as described here implements a restricted form of theory revision, in which revisions are restricted to completing definitions of background predicates other than those for which observations are provided. This assumes that the background knowledge is correct, but incomplete. In general, if background predicates are both incorrect and incomplete, then a more elaborate procedure would be required.

3.10 Feature Construction

One promising role for ILP is in the area of feature construction. A good review of the use of ILP for this can be found in S. Kramer, N. Lavrac and P. Flach (2001), *Propositionalization Approaches to Relational Data Mining*, in Relational Data Mining, S. Dzeroski and N. Lavrac (eds.), Springer.

Aleph uses a simple procedure to construct boolean features. The procedure is invoked using the `induce_features/0` command. This is almost identical to the `induce_cover/0` command. Recall that `induce_cover/0` uses a a covering strategy to construct rules that explain the examples (the slight twist being that all positive examples are retained when evaluating clauses at any given stage). The difference with `induce_features/0` is that all good clauses that are found during the course of constructing such rules are stored as new features. A feature stored by Aleph contains two bits of information: (1) a number, that acts as a feature identifier; and (2) a clause `(Head:-Body)`. Here `Head` is a literal that unifies with any of the examples with the same name and arity as `Head` and `Body` is a conjunction of literals. The intent is that the feature is `true` for an example if and only if the example unifies with `Head` and `Body` is `true`. For classification problems, the user has to specify the the dependent variable. This is done using `set(dependent,...)`.

The process of finding rules (and the corresponding features) continues until all examples are covered by the rules found or the number of features exceeds a pre-defined upper limit (controlled by `set(max_features,...)`).

What constitutes a "good clause" is dictated by settings for various Aleph parameters. The following settings are an example of some parameters that are relevant:

```
:- set(clauselength,10).
:- set(minacc,0.6).
:- set(minscore,3).
:- set(minpos,3).
:- set(noise,50).
:- set(nodes,5000).
:- set(explore,true).
:- set(max_features,20000).
```

Features found by Aleph can be shown by the `show(features)` command. Aleph can be used to show the boolean vectors for the train and test examples using a combination of `set(portray_examples,...)`, `features/2` appropriate definitions for `aleph_portray/1` and `show(train_pos)`, `show(train_neg)` etc. Here is an example of the use of `aleph_portray/1` for examples in the training set:

```
                    aleph_portray(train_pos):-
                            setting(train_pos,File),
    show_features(File,positive).
                    aleph_portray(train_neg):-
                            setting(train_neg,File),
    show_features(File,negative).
```

```
                    show_features(File,Class):-
                            open(File,read,Stream),
                            repeat,
     read(Stream,Example),
                            (Example = end_of_file -> close(Stream);
                                    write_features(Example,Class),
                                    fail).

                    write_features(Example,_):-
                            features(_,(Example:- Body)),
                            (Body -> write(1), write(' '); write(0), write(' ')),
                            fail.
                    write_features(_,Class):-
                            writeq(Class), nl.
```

If `portray_examples` is set to `true`, Aleph will call `aleph_portray(Term)`, when the command `show(Term)` is executed (with `Term` being one of `train_pos`, `train_neg`, `test_pos` or `test_neg`).

3.11 Other commands

There are a number of other useful commands in Aleph. These are:

`rdhyp` Read a hypothesised clause from the user.

`addhyp` Add current hypothesised clause to theory. If a search is interrupted, then the current best hypothesis will be added to the theory.

`sphyp` Perform Generalised Closed World Specialisation (GCWS) on current hypothesis. This can result in the creation of new abnormality predicates which define exceptional conditions (see Chapter 5 [Notes], page 41)

`addgcws` Add hypothesis constructed by performing GCWS to theory.

`covers` Show positive examples covered by hypothesised clause.

`coversn` Show negative examples covered by hypothesised clause.

`reduce` Run a search on the current bottom clause, which can be obtained with the `sat/1` command.

`man(-`*V*`)` *V* is of location of the on-line manual.

`abducible(+`*V*`)`
V is of the form *N/A*, where the atom *N* is the name of the predicate, and *A* its arity. Specifies that ground atoms with symbol *N/A* can be abduced if required.

`commutative(+`*V*`)`
V is of the form *N/A*, where the atom *N* is the name of the predicate, and *A* its arity. Specifies that literals with symbol *N/A* are commutative.

`symmetric(+V)`
: V is of the form N/A, where the atom N is the name of the predicate, and A its arity. Specifies that literals with symbol N/A are symmetric.

`lazy_evaluate(+V)`
: V is of the form N/A, where the atom N is the name of the predicate, and A its arity. Specifies that outputs and constants for literals with symbol N/A are to be evaluated lazily during the search. This is particularly useful if the constants required cannot be obtained from the bottom clause constructed by using a single example. During the search, the literal is called with a list containing a pair of lists for each input argument representing 'positive' and 'negative' substitutions obtained for the input arguments of the literal. These substitutions are obtained by executing the partial clause without this literal on the positive and negative examples. The user needs to provide a definition capable of processing a call with a list of list-pairs in each argument, and how the outputs are to be computed from such information. For further details see A. Srinivasan and R. Camacho, *Experiments in numerical reasoning with ILP*, To appear: Jnl. Logic Programming.

`model(+V)`
: V is of the form N/A, where the atom N is the name of the predicate, and A its arity. Specifies that predicate N/A will be used to construct and execute models in the leaves of model trees (see Section 3.6 [Tree Learning], page 28). This automatically results in predicate N/A being lazily evaluated (see `lazy_evaluate/1`).

`positive_only(+V)`
: V is of the form N/A, where the atom N is the name of the predicate, and A its arity. States that only positive substitutions are required during lazy evaluation of literals with symbol N/A. This saves some theorem-proving effort.

`random(V,+D)`
: V is a random variable from distribution D. D is the specification of a discrete or normal distribution. The discrete distribution is specified as [p1-a,p2-b,...] where "p1" represents the probability of drawing element "a", "p2" the probability of drawing element "b" and so on. A normal distribution with mean "m" and standard deviation "s" is specified by the term "normal(m,s)". If V is bound to a value then the predicate succeeds if and only if the value has a non-zero probability of occurrence (which is trivially satisfied for a normal distribution).

`sat(+V)`
: V is an integer. Builds the bottom clause for positive example number V. Positive examples are numbered from 1, and the numbering corresponds to the order of appearance in the `.f` file.

`example_saturated(-V)`
: V is a positive example. This is the current example saturated.

`show(+V)`
: Different values of V result in showing the following.

  `bottom` Current bottom clause.

`constraints`	Constraints found by `induce_constraints`.
`determinations`	Current determination declarations.
`features`	Propositional features constructed from good clauses found so far.
`gcws`	Hypothesis constructed by the `gcws` procedure.
`good`	Good clauses found in searches conducted so far (good clauses all have a utility above that specified by `minscore`).
`hypothesis`	Current hypothesised clause.
`modes`	Current mode declarations (including all modeh and modeb declarations).
`modehs`	Current modeh declarations.
`modebs`	Current modeb declarations.
`neg`	Current negative examples.
`pos`	Current positive examples.
`posleft`	Positive examples not covered by theory so far.
`rand`	Current randomly-generated examples (used when `evalfn` is `posonly`).
`search`	Current search (requires definition for `portray(search)`).
`settings`	Current parameter settings.
`sizes`	Current sizes of positive and negative examples.
`theory`	Current theory constructed.
`test_neg`	Examples in the file associated with the parameter `test_neg`.
`test_pos`	Examples in the file associated with the parameter `test_pos`.
`train_neg`	Examples in the file associated with the parameter `train_neg`.
`train_pos`	Examples in the file associated with the parameter `train_pos`.
`Name/Arity`	Current definition of the predicate Name/Arity.

`redundant(+`*`Clause`*`,+`*`Lit`*`)`

A user-specified predicate that defines when a literal `Lit` is redundant in a clause `Clause`. `Clause` can be the special term `bot`, in which case it refers to the current bottom clause. Calls to this predicate are only made if the flag `check_redundant` is set to `true`.

`modeh(+`*`Recall`*`,+`*`Mode`*`)`
: *Recall* is one of: a positive integer or `*`. *Mode* is a mode template as in a `mode/2` declaration. Declares a mode for the head of a hypothesised clause. Required when `evalfn` is `posonly`.

`modeb(+`*`Recall`*`,+`*`Mode`*`)`
: *Recall* is one of: a positive integer or `*`. *Mode* is a mode template as in a `mode/2` declaration. Declares a mode for a literal in the body of a hypothesised clause.

`text(+`*`L`*`,+`*`T`*`)`
: *L* is a literal that can appear in the head or body of a clause. *T* is a list of terms that contain the text to be printed in place of the literal. Variables in the list will be co-referenced to variables in the literal. For example, `text(active(X),[X, 'is active'])`. Then the clause `active(d1)` will be written as `d1 is active`.

`hypothesis(-`*`Head`*`,-`*`Body`*`,-`*`Label`*`)`
: *Head* is the head of the current hypothesised clause. *Body* is the body of the current hypothesised clause. *Label* is the list `[P,N,L]` where `P` is the positive examples covered by the hypothesised clause, `N` is the negative examples covered by the hypothesised clause, and `L` is the number of literals in the hypothesised clause,

`feature(+`*`Id`*`,+`*`(Head:-Body)`*`)`
: Declares a new feature. *Id* is a feature identifier (usually a number). *Head* is a literal that can unify with one or more of the examples. *Body* is a conjunction of literals that constitutes the feature.

`features(?`*`Id`*`,?`*`(Head:-Body)`*`)`
: Checks for an existing feature. *Id* is a feature identifier (usually a number). *Head* is a literal that can unify with one or more of the examples. *Body* is a conjunction of literals that constitutes the feature.

4 Related versions and programs

With appropriate settings, Aleph can emulate some the functionality of the following programs: P-Progol, CProgol, FOIL, FORS, Indlog, MIDOS, SRT, Tilde and WARMR. Descriptions and pointers to these programs are available at:

`http://www-ai.ijs.si/~ilpnet2/systems/`

In addition the following programs and scripts are relevant.

`T-Reduce`
: T-Reduce is a companion program to Aleph that can be used to process the clauses found by the commands `induce_cover` and `induce_max`. This finds a subset of these clauses that explain the examples adequately, and have lesser overlap in coverage. T-Reduce uses the **Yap** Prolog compiler. A copy of this program is available (without support) at:

  `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/treduce.pl`

  This has not been used for several years and is vulnerable to the usual forces of decay that afflict old programs.

`GUI`
: A graphical user interface to Aleph has been developed by J. Wielemaker and S. Moyle. This is written for SWI-Prolog and uses the XPCE library. Details of this can be obtained from S. Moyle (sam at comlab dot ox dot ac dot uk).

`Scripts`
: There are some scripts available for performing cross-validation with Aleph. Here is a copy of a Perl script written by M. Reid (mreid at cse dot unsw dot edu dot au):

  `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/xval_pl.txt`

  S. Konstantopoulos (konstant at let dot rug dot nl) and colleagues have a shell script and a Python script for the same purpose. Copies of these are at:

  `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/xval_sh.txt`

  and

  `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/xval_py.txt`

5 Notes

This section contains ideas and suggestions that have surfaced during the development of Aleph and its predecessor programs. The topics themselves are in no particular order. They are written in a somewhat stylised manner and reflect various personal biases. They should therefore, not be considered normative in any way.

5.1 On the appropriateness of Aleph

1. There are many ILP programs. Aleph is not particularly special.
2. Check whether the problem needs a relational learning program. Is it clear that statistical programs, neural networks, Bayesian nets, tree-learners etc. are unsuitable or insufficient?
3. Aleph's emulation of other systems is at the "ideas" level. For example, with a setting of `search` to `heuristic`, `evalfn` to `compression`, `construct_bottom` to `saturation`, and `samplesize` to `0`, the command `induce` will a construct a theory along the lines of the Progol algorithm described by S. Muggleton. This is, however, no substitute for the original. If you want an implementation of S. Muggleton's Progol algorithm exactly as described in his paper, then Aleph is not suitable for you. Try CProgol instead. The same comment applies to other programs listed in Chapter 4 [Other Programs], page 39.
4. Aleph is quite flexible in that it allows customisation of search, cost functions, output-display etc. This allows it to approximate the functionality of many other techniques. It could also mean that it may not be as efficient as special-purpose implementations. See also:

   `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/ilp_and_aleph.ps`

5.2 On predicate-name clashes with Aleph

1. You may get into trouble if predicate names in the background knowledge clash with those already used within Aleph. This may be benign (for example, two different predicates that encode the same relation) or malignant (with predicates that have the same name encoding quite different things). The list of predicate names already in use can be obtained by repeated calls to the `current_predicate(X)` goal provided by the Prolog engine.
2. It would be better if Aleph predicates were renamed, or some modular approach was adopted. None of this is done so far.

5.3 On the role of the bottom clause

1. Besides it's theoretical role of anchoring one end of the search space, the bottom clause is really useful to introduce constants (these are obtained from the seed example), and variable co-references.
2. If you are not interested in particular constants or the bottom clause introduces too many spurious co-references, it may be better not to construct a bottom clause. Try using the automatic refinement operator, or write your own refinement operator.

3. If the bottom clause is too large (> 500 literals), then simply printing it on screen takes a long time. Turn this off with setting verbosity to `0`.
4. If the bottom clause is too large (> 500 literals), then you can construct it lazily (during the search) by setting the `construct_bottom` flag to `reduction`.

5.4 On using Aleph interactively.

1. It is always worth experimenting with Aleph before constructing a full theory. The commands `sat/1` or `rsat/0`, followed by the command `reduce/0` are useful for this. `sat(N)` constructs the bottom clause for example number `N`. `rsat` constructs a bottom clause for a randomly selected example. `reduce` does a search for an acceptable clause.
2. You can interrupt a search at any time. The command `addhyp/0` then adds the current best clause to the theory. This has the flavour of anytime-learning.
3. The `induce_incremental` command is highly interactive. It requires the user to provide examples, and also categorise the result of searches. This may prove quite demanding on the user, but has the flavour of the kind of search done by a version-space algorithm.
4. Setting `interactive` to `true` and calling `induce_clauses` has the same effect as calling `induce_incremental`. Trees can also be constructed interactively by setting `interactive` to `true` and calling `induce_tree`.

5.5 On different ways of constructing a theory

1. The routine way of using `induce/0` is often sufficient.
2. `induce/0`, `induce_cover/0`, `induce_max/0`, `induce_clauses/0` and `induce_incremental/0` encode control strategies for clause-level search. They will use any user defined refinement operators, search and evaluation functions, beam-width restrcitions etc that are set. In terms of speed, `induce/0` is usually faster than `induce_cover/0`, which in turn is faster than `induce_max/0`. The time taken by `induce_incremental/0` is not as easily characterisable. `induce_clauses/0` is simply `induce/0` or `induce_incremental/0` depending on whether the flag `interactive` is `false` or `true` respectively.
3. `induce_max/0` results in a set of clauses that is invariant of example ordering. Neither `induce_cover/0`, `induce/0` or `induce_incremental/0` have this property.
4. Use the T-Reduce program after `induce_max/0` or `induce_cover/0` to obtain a compact theory for prediction.
5. You can construct a theory manually by repeatedly using `sat/1` (or `rsat/0`), `reduce/0` and `addhyp/0`.
6. You can mitigate the effects of poor choice of seed example in the saturation step by setting the `samplesize` flag. This sets the number of examples to be selected randomly by the `induce` or `induce_cover` commands. Each example seeds a different search and the best clause is added to the theory.
7. If you set `samplesize` to `0` examples will be selected in the order of appearance in the positive examples file. This will allow replication of results without worrying about variations due to sampling.

8. The `induce_tree` command will construct tree-structured theories.
9. The `induce_theory` command is to be used at your own peril.

5.6 On a categorisation of parameters

1. The following parameters can affect the size of the search space: `i`, `clauselength`, `nodes`, `minpos`, `minacc`, `noise`, `explore`, `best`, `openlist`, `splitvars`.
2. The following parameters affect the type of search: `search`, `evalfn`, `refine`, `samplesize`.
3. The following parameters have an effect on the speed of execution: `caching`, `lazy_negs`, `proof_strategy`, `depth`, `lazy_on_cost`, `lazy_on_contradiction`, `searchtime`, `prooftime`.
4. The following parameters alter the way things are presented to the user: `print`, `record`, `portray_hypothesis`, `portray_search`, `portray_literals`, `verbosity`,
5. The following parameters are concerned with testing theories: `test_pos`, `test_neg`, `train_pos`, `train_neg`.

5.7 On how the single-clause search is implemented

1. The search for a clause is implemented by a restricted form of a general branch-and-bound algorithm. A description of the algorithm follows. It is a slight modification of that presented by C.H. Papadimitriou and K. Steiglitz (1982), *Combinatorial Optimisation*, Prentice-Hall, Edgewood-Cliffs, NJ. In the code that follows, *activeset* contains the set of "live" nodes at any point; the variable *C* is used to hold the cost of the best complete solution at any given time.

```
begin
    active:= {0}; (comment: ``0'' is a conventional starting point)
    C:= inf;
    currentbest:= anything;
    while active is not empty do begin
        remove first node k from active; (comment: k is a branching node)
        generate the children i=1,...,Nk of node k, and
            compute corresponding costs Ci and
                lower bounds on costs Li;
        for i = 1,...,Nk do
            if Li >= C then prune child i
            else begin
                if child i is a complete solution and Ci < C then begin
                        C:= Ci, currentbest:= child i;
                        prune nodes in active with lower bounds more than Ci
                end
                add child i to active
            end
    end
end
```

2. The algorithm above results in a search tree. In Aleph, each node contains a clause.

3. A number of choices are made in implementing a branch-and-bound algorithm for a given problem. Here are how these are made in Aleph: (a) *Branch node.* The choice of node to branch on in the activeset is based on comparisons of a dual (primary and secondary) search key associated with each node. The value of this key depends on the search method and evaluation function. For example, with `search` set to `bf` and `evalfn` set to `coverage` (the default for Aleph), the primary and secondary keys are `-L,P-N` respectively. Here `L` is the number of literals in the clause, and `P,N` are the positive and negative examples covered by the clause. This ensures clauses with fewer literals will be chosen first. They will further be ordered on difference in coverage; (b) *Branch set.* Children are generated by refinement steps that are either built-in (add 1 literal at a time) or user-specified. With built-in refinement, loop detection is performed to prevent duplicate addition of literals; (c) *Lower bounds.* This represents the lowest cost that can be achieved at this node and the sub-tree below it. This calculation is dependent on the search method and evaluation function. In cases where no easy lower bound is obtainable, it is taken as `0` resulting in minimal pruning; (d) *Restrictions.* The search need not proceed until activeset is empty. It may be terminated prematurely by setting the `nodes` parameter. Complete solutions are taken to be ones that satisfy the language restrictions and any other hypothesis-related constraints.

5.8 On how to reduce the search space

1. Use smaller `i` setting or smaller `clauselength` or `nodes` setting. Avoid setting `splitvars` to `true` (it is not even clear whether this works correctly anyway). Try relaxing `minacc` or `noise` to allow clauses with lower accuracy. Set `minpos` to some larger value than the default. Set a different value to `best`.
2. Write constraints and prune statements.
3. Use a refinement operator that enumerates a smaller space.
4. Restrict the language by allowing fewer determinations.
5. Restrict the search space by setting beam-width (using parameter `openlist`); or using an iterative beam-width search (setting `search` to `ibs`); or using randomised local search (setting `search` to `rls`) with an appropriate setting for associated parameters); or using Camacho's language search (using parameter `language` or setting `search` to `ils`).
6. Use a time-bounded search by setting `searchtime` to some small value.

5.9 On how to use fewer examples

1. It need not be necessary to test on the entire dataset to obtain good estimates of the cost of a clause.
2. Methods like sub-sampling or windowing can be incorporated into ILP programs to avoid examining entire datasets. These are not yet incorporated within Aleph, although windowing can be achieved within a general purpose theory-revision program called **T-Revise** which can use any ILP program as its generalisation engine (available from Ashwin Srinivasan, ashwin at comlab dot ox dot ac dot uk). More details on this are available in: A. Srinivasan (1999), *A study of two sampling methods for analysing large datasets with ILP*, Data Mining and Knowledge Discovery, 3(1):95-123.

3. Using the `posonly` evaluation function will allow construction of theories using positive examples only (thus, some savings can be made by ignoring negative examples).

5.10 On a user-defined view of hypotheses and search

1. User-definitions of `portray/1` provide a general mechanism of altering the view of the hypotheses and search seen by the user.
2. There are 3 flags that are used to control portrayal. These are `portray_hypothesis`, `portray_search` and `portray_literals`. If the first is set to `true` then the command `show(hypothesis)` will execute `portray(hypothesis)`. This has to be user-defined. If the second flag is set to `true` then the command `show(search)` will execute `portray(search)`. This has to be user-defined. If the third flag is set to `true` then any literal `L` in a clause constructed during the search will be shown on screen by executing `portray(L)`. This has to be user-defined.
3. Examples of using these predicates can be found in the `portray` sub-directory in:

   `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

5.11 On numerical reasoning with Aleph

1. There are many programs specialised to accomplish numerical reasoning. Aleph is not one of them. Consider parametric techniques, regression trees etc. The ILP program **FORS** is an example of an ILP program particularly suited to perform regression like tasks (see A. Karalic and I. Bratko (1997), *First-Order Regression*, Machine Learning, 26:147-176). The program **SRT** is a first-order variant of a regression tree builder (see S. Kramer (1996), *Structural Regression Trees*, Proc. of the 13th National Conference on Artificial Intelligence (AAAI-96)), and the program **Tilde** has the capability of performing regression-like tasks (see H. Blockeel, L. De Raedt and J. Ramon (1998), *Top-down induction of clustering trees*, Proc of the 15th International Conference on Machine Learning, pp 55-63). Aleph does have a simple tree-based learner that can construct regression trees (see Section 3.6 [Tree Learning], page 28).
2. It is possible to attempt guesses at numerical constants that add additional literals to the bottom clause. An example of how this can be done with a predicate with multiple recall is in the Aleph files `guess.b`, `guess.f`, and `guess.n` in the `numbers` sub-directory in:

   `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`
3. Guessing may not always work. The problem may then be amenable to the use of the technique of lazy evaluation. Here an appropriate constant in literal Li is obtained during the search by calling a definition in background knowledge that calculates the constant by collecting bindings from pos examples that are entailed by the ordered clause L0, L1, ... Li-1, and the neg examples inconsistent with the ordered clause L0, L1, ... Li-1 (ie the pos and neg examples "covered" by this clause). An example of how this can be done is in the Aleph files `ineq.b`, `ineq.f`, and `ineq.n` in the `numbers` sub-directory in:

`http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

4. The technique of lazy evaluation can be used with more than one input argument and to calculate more than one constant. With several input arguments, values in lists of substitutions can be paired off. An example where it is illustrated how a line can be constructed from a picking two such substitution-pairs can be found in the Aleph files `ineq.b`, `ineq.f`, and `ineq.n` in the `numbers` sub-directory in:

   `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/examples.zip`

5. The use of lazy evaluation, in combination with user-defined search specifications can result in quite powerful (and complex) clauses. In the file:

   `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/mut.b`

   is the background knowledge used to construct theories in a subset of the "mutagenesis" problem. It illustrates the call to a C function to compute linear regression, user-defined refinement operators, and a user-defined cost function that forces clauses to be scored on mean-square -error (rather than coverage)

5.12 On applications of Aleph

1. Earlier incarnations of Aleph (called P-Progol) have been applied to a number of real-world problems. Prominent amongst these concern the construction of structure-activity relations for biological activity. In particular, the results for mutagenic and carcinogenic activity have received some attention. Also prominent has the been the use for identifying pharmacophores – the three-dimensional arrangement of functional groups on small molecules that enables them to bind to drug targets. See:

   `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/applications.html`.

2. Applications to problems in natural language processing have been done by James Cussens and others. See:

   `http://www.cs.york.ac.uk/~jc/`

5.13 On using Aleph with other techniques

1. There is often a significant advantage in combine the results of Aleph with those of established prediction methods.
2. Three ways of doing this are evident: (a) *As background knowledge.* Incorporate other prediction methods as part of the background knowledge for Aleph. An example is the use of linear regression as a background knowledge; (b) *As new features.* Incorporate the results from Aleph into an established prediction method. An example is the conversion of Aleph derived alerts into "indicator" variables for linear regression; and (c) *For outlier analysis.* Use Aleph to explain only those instances that are inadequately modelled by established techniques. An example is the use of Aleph to explain the non-linearities left after the linear component adequately explained by regression is removed.

5.14 On performing closed-world specialisation with Aleph

1. Generalised Closed-World Specialisation (GCWS) is a way of obtaining structured theories in ILP. Given an overgeneral clause C, GCWS specialises it by constructing automatically new "abnormality" predicates that encode exceptions to C, exceptions to those exceptions, etc.
2. A classic example is provided by the Gregorian Calendar currently in use in parts of the world. From 45 B.C.E to 1581 C.E the Holy Roman Empire subscribed to the Julian calendar commissioned by Julius Caesar. This specified that every year that was a multiple of 4 would contain an intercalary day to reconcile the calendar with a solar year (that is, one extra day would be added). This rule is correct up to around one part in a hundred and so up until 1582 errors could simply be treated as noise. In 1582 C.E Pope Gregory XIII introduced the Gregorian calendar. The following corrections were implemented. Every fourth year would be an intercalary year except every hundredth year. This rule was itself to be overruled every four hundredth year, which would be an intercalary year. As a set of clauses the Gregorian calendar is:

   ```
   normal(Y):-
           not(ab0(Y)).

   ab0(Y):-
           divisible(4,Y),
           not(ab1(Y)).

   ab1(Y):-
           divisible(100,Y),
           not(ab2(Y)).

   ab2(Y):-
           divisible(400,Y).
   ```

   where `normal` is a year that does not contain an intercalary day. With background knowledge of `divisible/2` GCWS would automatically specialise the clause:

   ```
   normal(Y).
   ```

   by constructing the more elaborate theory earlier. This involves invention of the `ab0,ab1,ab2` predicates.
3. See M. Bain, (1991), *Experiments in non-monotonic learning*, Eighth International Conference on Machine Learning, pp 380-384, Morgan Kaufmann, CA; and A. Srinivasan, S.H. Muggleton, and M. Bain (1992): *Distinguishing Noise from Exceptions in Non-Monotonic Learning*, Second International Workshop on ILP, for more details of GCWS.
4. The way to use GCWS within Aleph is as follows. First try to learn a clause in the standard manner (that is using the `sat` and `reduce` commands). If no acceptable clause is found, decrease the minimum accuracy of acceptable clauses (by setting `minacc` or `noise`). Now do the search again. You will probably get an overgeneral clause (that is, one that covers more negative examples than preferrable). Now use the `sphyp` command to specialise this hypothesis. Aleph will repeatedly create examples for new

abnormality predicates and generalise them until the original overgeneral clause does not cover any negative examples. You can then elect to add this theory by using the `addgcws` command.

5. The implementation of GCWS within Aleph is relatively inefficient as it requires creating new examples for the abnormality predicates on disk.

5.15 On some basic ideas relevant to ILP

1. Some basic ideas relevant ILP can be found at:

   `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/basic.html`

6 Change Logs

6.1 Changes in Version 1

- **Wed Nov 10 10:15:44 GMT 1999:** fixed bug in bug fix of Fri Oct 8 10:06:55 BST 1999.
- **Mon Oct 25 14:06:07 BST 1999:** minor improvement to code for stochastic clause selection; added mailing list info in header
- **Fri Oct 8 10:06:55 BST 1999:** fixed bug in `record_testclause` to add depth bound call to body literals.
- **Mon Sep 20 09:50:23 BST 1999:** fixed bug in `continue_search` for user defined cost function; fixed bug in stochastic clause selection that attempts to select more literals than present in bottom clause.

6.2 Changes in Version 2

- **Fri Mar 31 17:12:52 BST 2000:** Some predicates called during variable-splitting did not account for change that allows arbitrary terms in mode declarations. Changed split_args/4 to split_args/5 to fix bug concerning multiple modes for the same predicate.
- **Thu Mar 23 09:57:15 GMT 2000:** Minor fixes. Some predicates called during lazy evaluation did not account for change that allows arbitrary terms in mode declarations.
- **Fri Jan 28 14:57:32 GMT 2000:** Arbitrary terms now allowed in mode declarations; logfile no longer records date of trace automatically (a system call to 'date' causes Yap to crash on some non-Unix systems – use set(date,...) to record date).

6.3 Changes in Version 3

- **Wed May 16 06:22:52 BST 2001:**
  - Changed retractall to retract_all
  - Added check for setting(refine,user) in check_auto_refine (reported by Khalid Khan)
  - Added clause to select_nextbest/2 for RefineType = user
  - Fixed call to get_gains in reduce(_) to include StartClause when using a refinement operator
  - Some calls to idb entries for last_refinement and best_refinement were incorrectly using key: `"search"` instead of `"aleph"`
  - Clause in get_refine_gain and get_refine_gain1 when RefineType \= rls had variable name clash for variable E. Renamed one of these to Example
  - Changed representation of gains for openlist. This is now the term [P|S] where P is the primary key and S is the secondary key. This used to be converted into a unique number, which required the setting of a base. This is no longer required and so removed fix_base predicate. Corresponding changes to structure for gains idb also implemented by including P and S as first two arguments, and to uniq_insert to compare using lexicographically
  - Call to reduce now catches aborts and reinstates the values of any parameters saved via the use of catch/3

  - Rls search type is now correctly heuristic (and not bf: reported by David Page and Khalid Khan)
  - Incorporated Filip Zelezny's corrections to posonly estimate by ensuring that rm_seeds updates atoms_left for rand examples.
  - sample_clauses can now use probability distribution over clauselengths
  - Reinstated search 'scs' to perform stochastic clause selection (after Aleph 0 this was being done as a special case of rls before)
  - Removed call to store_cover to fix problem identified by Stasinos Konstantopoulos when get_hyp_label/2 calls covers/1 and coversn/1
  - Updated manual to mirror style of Yap's manual and added patches sent by Stasinos Konstantopoulos
- **Fri May 18 07:44:02 BST 2001:** Yap was unable to parse calls of the form recorded(openlist,[[K1|K2]|_],_) (reported by Khalid Khan). Worked around by changing to recorded(openlist,[H|_],_), H= [K1|K2].
- **Wed Jul 25 05:50:12 BST 2001:**
  - Changed calls to val(ArgNo,Pos). This was causing variable-splitting to fail
  - Both input and output variables can now be split in the head literal
  - Posonly learning now adds an SLP generator clause for each modeh declaration
  - Modes can now contain ground terms
  - Restored proper operation of user-defined refinement operator
  - Added facility for time-restricted proofs
  - Added facility for new computation rule that selects leftmost literals with delaying
- **Mon Mar 18 12:49:10 GMT 2002:**
  - Changed update atoms/2 to check the mode of the ground literal used to produce the bottom clause. This means copies of ground literals can now exist, if the corresponding variables are typed differently by the mode declarations. This was prompted by discussions with Mark Rich.
  - continue_search/3 replaced by discontinue_search/3.
  - Added setting for newvars. This bounds the maximum number of new variables that can be introduced in the body of a clause
  - Added code developed by Filip Zelezny to implement randomised local search using 'randomised rapid restarts'.
  - Changed pos_ok/6 to check for minpos constraint for any refinement operator r that is such that for a hypothesis H, poscover(r(H)) <= poscover(H). This cannot be guaranteed when search = rls or refine = user. In other situations, the built-in refinement operator that adds literals is used and this property is holds. This was prompted by discussions with James Cussens.
  - Fixed bug in randomised search: rls_nextbest/4 that had args for gain/4 in the wrong order.
  - Fixed closed-world specialisation: was not checking for lazy evaluation, also changed tmp file names to alephtmp.[fn]
  - subsumes/2 renamed aleph_subsumes/2.

- Changes to lazy evaluation code that allows a set of input bindings from an example. This makes multi-instance learning possible.
- Automatic general-to-specific refinement from modes now ensures that it does not generate clauses that would succeed on prune/1 .
- Built-in local clause moves in randomised search now ensures that it does not generate clauses that would succeed on prune/1 .
- Random sampling of clauses from hypothesis space now returns most general clause on failure.
- Added code check_recursive_calls/0. This allows calls to the positive examples when building the bottom clause if recursion is allowed.
- Changed covers/1 and and coversn/1 to check if being called during induce/0.
- Miscellaneous changes of write/1 to writeq/1.

6.4 Changes in Version 4

- **Wed Nov 13 16:18:53 GMT 2002:**
  - Added portability to SWI-Prolog.
  - Lazy-evaluation now creates literals identified by numbers that are less than 0 (rather than by positive numbers beyond that obtained from the bottom clause).
  - Fixed error in mark_redundant_lits/2 that checked for redundant literals in the bottom clause.
  - Avoided overloading of the refine flag by introducing a secondary flag refineop that is actually used by Aleph.
  - Avoided overloading of the search flag by introducing a secondary flag searchstrat that is actually used by Aleph.
  - Removed defunct flags verbose, computation_rule.
  - Added code symmetric_match/2 when checking for symmetric literals.
  - Added new flags including minposfrac, minscore, mingain, prune_tree, confidence, classes, newvars etc.
  - Changed flags so that noise/minacc can co-exist. Now the user's problem to check that these are consistent.
  - Introduced new predicate find_clause/1 to perform basic searches (this was previously done by reduce/1).
  - Miscellaneous rewrites of code for checking lang_ok and newvars_ok.
  - Miscellaneous rewrites of code for optimising clauses.
  - Rationalised pruning code.
  - Fixed bug in pos_ok that affected posonly mode.
  - Added code for dealing uniformly with plus and minus infinities in SWI and Yap.
  - Added code for dealing uniformly with alarms in SWI and Yap.
  - Added code for dealing uniformly with random number generation in SWI and Yap.
  - Added code for dealing with cputime in cygnus (from Mark Reid).